User manual for Uranie v4.10.0

C++ version

February 21, 2025

THE URANIE TEAM, support-uranie@cea.fr

Contents

List of Figures

List of Tables

Chapter I

Overview: Uranie in a nutshell

I.1 Introducing Uranie

Uranie (the version under discussion here being v4.10.0) is a software dedicated to perform studies on uncertainty propagation, sensitivity analysis and surrogate model generation and calibration, based on ROOT (the corresponding version being v6.32.08).

As a result, Uranie benefits from numerous features of ROOT, among which:

- an interactive C++ interpreter (Cling), built on the top of LLVM and Clang;
- a Python interface (PyROOT);
- an access to **SQL** databases;
- many advanced data visualisation features;
- and much more...

In the following sections, the ROOT platform will be briefly introduced as well as the python interface it brings once the Uranie classes are declared and known. The organisation of the Uranie platform is then introduced, from a broad scale, giving access to more refined discussion within this documentation.

I.1.1 Uranie modules organisation

The platform consists of a set of so-called **technical libraries**, or **modules** (represented as green boxes in Figure I.1), each performing a specific task.

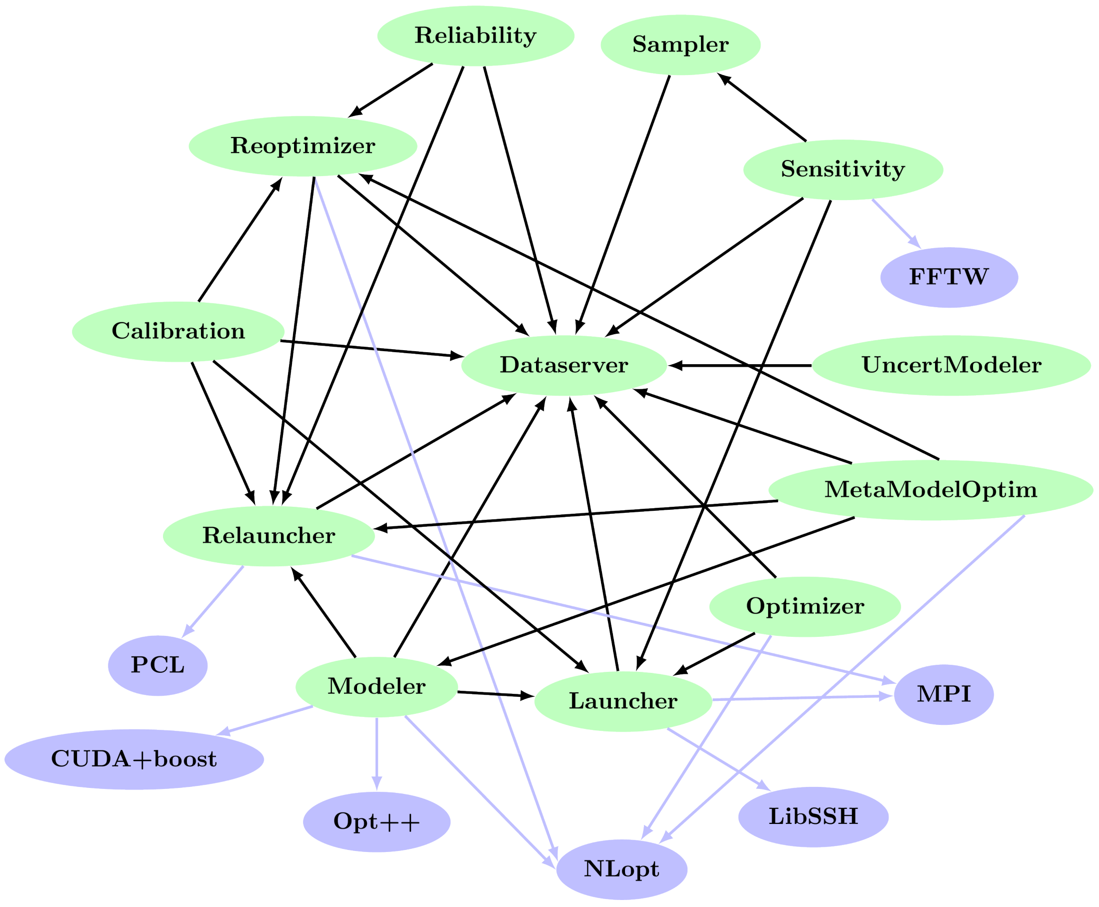


Figure I.1: Organisation of the Uranie-modules (green boxes) in terms of inter-dependencies. The blue boxes represent the external dependencies (discussed later on).

In the rest of this section each and every modules discussed in this documentation will be briefly described (in terms of role and main components) starting with the DataServer one, which is the spine of the Uranie project, as shown in Figure I.1. A more precise description will then be done in the dedicated other chapters.

I.1.1.1 DataServer module

The **DataServer** library (cf Chapter II) is the core of the Uranie platform. It describes the central element of Uranie: the `TDataServer`. This object contains all the necessary information about the variables of a problem (such as the names, units, probability laws, data files, and so on...) and allows to perform the very basic statistical operations.

I.1.1.2 Sampler module

The **Sampler** library (cf Chapter III) allows to create design-of-experiments using `TDataServer`'s attributes which are *random variables*. There are a large variety of design-of-experiments some of which are only meant to be called by more complicated methods.

I.1.1.3 Launcher module

The **Launcher** library (cf Chapter IV) applies an analytic function or an external simulation code on the content of a `TDataServer`. The `TDataServer` content can either result from a design-of-experiments generated using one of the `TSampler` object, or can be loaded into from an external source (*ASCII* file, SQL database, *etc*...).

I.1.1.4 Modeler module

The **Modeler** library (cf Chapter V) allows the construction of a *surrogate model* that links the output Y and the input factors X_i (for $i = 1, \ldots, n_X$) (polynomial models, neural networks, ...).

I.1.1.5 Sensitivity module

The **Sensitivity** library (cf Chapter VI) allows to perform sensitivity analysis of one of the output response Y with respect to the input factors X_i (for $i = 1, \ldots, n_X$). The very basic concepts of sensitivity analysis are introduced as well in the introduction of this chapter while they are discussed a bit more thoroughly in [30].

I.1.1.6 Optimizer and Reoptimizer modules

The **Optimizer** and **Reoptimizer** libraries (cf Chapter VII and Chapter IX) are dedicated to optimisation and model calibration. Model calibration consists in setting up the "degrees of freedom" of a model so that future simulations will optimally fit an experimental database. The optimisation is a complex procedure and several techniques are available to perform single-criterion or multi criteria one, with and without constraint.

I.1.1.7 Relauncher module

The **Relauncher** library (cf Chapter VIII) is more a technical module that is used throughout the Uranie platform to provide a general architecture for all parametric studies, allowing secure multi-processors and multi-thread usage in a simple way.

I.1.1.8 Calibration module

The **Calibration** library (cf Chapter XI) is more a dedicated module that is used to get the best estimations of some of the parameter of a specific model under consideration. This module provides different techniques relying on their own hypothesis on the model but all of these methods need data to perform this calibration.

I.1.2 External dependencies

Before starting with the internal organisation of the platform, this section is discussing the dependencies of the Uranie platform. They are sorted in two categories: the compulsory and optional ones. The latter ones are shown as blue boxes in Figure I.1 and both types are listed and briefly discussed below.

I.1.2.1 Compulsory dependencies

ROOT: An oriented-object package that offers many possibilities for data handling, analysis, display... [31]

Sources, binaries and documentation are available at http://root.cern.ch (version used is v6.32.08)

Cmake: Free and open-source software for managing the build process of compiled software [32].

It is available at http://www.cmake.org/ (version used is v3.28.3)

CPPUnit: Unit testing framework for C++ programming [37].

It is available at http://sourceforge.net/projects/cppunit/ (version used is v1.15.1)

I.1.2.2 Optional dependencies

OPT++: Libraries that include non linear optimisation algorithms written in C++ [33].

It is available at https://software.sandia.gov/opt++/ (version used is v2.4). It is mainly used for neural networks.

FFTW: Library that computes discrete Fourier transform (DFT) (one or more dimensions), of arbitrary input size [34].

It is available at http://www.fftw.org/ (version used is v3.3.10). It is mainly used for two methods in the Sensitivity library.

NLopt: Library for nonlinear optimisation [35].

It is available at http://ab-initio.mit.edu/nlopt (version used is v2.7.1). It is mainly used for kriging and mono-criterion optimisation.

Boost It is a set of libraries used here to implement the low level functionality for coroutines in replacement of PCL (Portable Coroutine Library) since V4.8.0.

It is available at https://www.boost.org (version used is v1.83)

MPI: (Message Passing Interface) Standardised and portable message-passing system needed to run parallel computing [36].

It is available at http://www.open-mpi.org/ (version used is v3.1)

CUDA: (Compute Unified Device Architecture) Parallel computing platform and pro- gramming model invented by NVIDIA to harness the power of the graphics processing unit (GPU) [38](version used is v8.0).

If requested, it should be used with the **boost** library, with a version greater than v1.83.

I.2 ROOT Environment

Uranie can be seen as a set of extension libraries for ROOT. As such, its traditional use is through the ROOT tools, in particular the `root` command which offers a C++ interpreter. This is presented in this section. In the next section, PyRoot will be presented.

I.2.1 Environment variables

Assuming you are in a ROOT enabled system, your shell environment probably defines `ROOTSYS`, `PATH` and `LD_LIBRARY_PATH` variables (unless they are installed in standard location). In order to use Uranie, assuming it is not installed in standard or ROOT location, the environment variable `LD_LIBRARY_PATH` needs to be completed and must contain the sub directory `lib` of the Uranie installation.

To achieve this, one can set the environment variable `URANIESYS` to Uranie's installation directory. Then, depending on the shell family used, the variable `LD_LIBRARY_PATH` can be updated as follows:

C Shell Family (csh, tcsh, etc.)

```
##------ Uranie ------
setenv URANIESYS MyUranieInstallDirectory

setenv LD_LIBRARY_PATH ${URANIESYS}/lib:$LD_LIBRARY_PATH
```

Bourne Shell Family (sh, bash, etc.)

```
##------ Uranie ------
export URANIESYS=MyUranieInstallDirectory

export LD_LIBRARY_PATH=${URANIESYS}/lib:$LD_LIBRARY_PATH
```

Uranie may need some external libraries that must be reachable via the `LD_LIBRARY_PATH`. You can adapt the former script to take it into account. Uranie installation procedure provides a configuration script that tries to create a safe environment.

I.2.2 ROOT interpreter and runtime compiler

Uranie can be used through the C++ interpreter of ROOT, either using the command line, or in **batch** mode, the successive commands being stored in a file called a **ROOT macro** .

To run the macro "`MyMacro.C`" from a shell prompt, use the command:

```
root -l MyMacro.C
```

The `-l` option prevents from launching the ROOT login window. Other options that may be useful are:

- `-q`: quit ROOT interpreter when the script is run.
- `-b`: run in batch mode (without DISPLAY)

These options are typically used when your job needs to be distributed.

To execute a Macro from the ROOT prompt, use the command:

```
.x MyMacro.C
```

ROOT interpreter offers some facilities for macro writing. In particular, it takes care of loading the needed libraries, and does not need the `#include` directives.

WARNING WITH ROOT6: The new versions of ROOT (from ROOT 6.0) is now using Cling the interpreter that does also a compilation on the fly thanks to LLVM and Clang. It is now much more compliant with the C++ syntax that previous version, allowing only few simplifications (some namespace loaded and include lines brought).

Another way to run a macro in a compile way is to append one or two "+" at the end of the script.

```
root -l MyMacro.C+
```

A "+" compiles the macro if the source changes, while "++" forces a recompilation. From ROOT 6, this method does not go faster that the "interpreted" one. The only difference remaining now is to produce easily a library (`.so`) file.

This runtime compiler still deals with libraries auto-load, but needs the `#include` directives. ROOT looks for include files in standard location and in the `$ROOTSYS/include` directory. If Uranie is installed elsewhere, you have to specify where to find its include files.

One way to do so is to use the `.rootrc` file. Here is an example:

.rootrc

```
ACLiC.IncludePaths: -DJIT -I$URANIESYS/include -I/another/include/path
```

It defines a list of C++ compiler options that are added at compile time: `-I` adds an include path; `-D` defines a CPP macro. This CPP macro can be used to write a single source code for both interpreted or compiled runs, hiding or showing part of source lines (`#include` declarations for example).

Another way is to use the `rootlogon.C` file. If this file is available in the active directory, it will be systematically loaded by ROOT before the application is launched. A more thorough discussion on this file can be found in Section I.2.4.

rootlogon.C

```
{
  gInterpreter->AddIncludePath("$URANIESYS/include")
  gInterpreter->AddIncludePath("/another/include/path")
}
```

This allows both the interpreter and the compiler to find the needed files.

One important advantage of runtime compilation is that error messages are more easily understandable.

I.2.3 Standard compilation

For C++ enthusiasts, it is possible to use ROOT (and subsequently Uranie) as a set of libraries, and to compile an executable. In this case, you have to take care of linked libraries. In order to help the user, ROOT provides a macro that give many of the needed flag to perform the compilation, both to give the path to the headers and the path to the libraries.

```
g++ -o OutputName TheFileName.C  `root-config --cflags --libs`
```

For this example:

- OutputName: the name of the resulting executable
- TheFileName.C: the C++ file containing the code
- `root-config --cflags --libs`: this command provides all the necessary flags to compile most of the macros using ROOT.

This is a logic we acknowledge and try to follow as well for Uranie: if one wants to compile an Uranie macro, one can use two flags.

- URANIECPPFLAG: it defines all the include path needed and add on top the ones from ROOT;
- URANIELDFLAG: it defines the linking option and path to library and add on top the ones from ROOT;

With this, an example of standalone compilation for a given macro `UranieMacro.C` will be:

```
g++ -o OutputExecutable UranieMacro.C  `echo ${URANIECPPFLAG} ${URANIELDFLAG}`
```

I.2.4 Uranie namespace

As for the vast majority of ROOT's classes, Uranie's classes started with a capital T (for instance "TFoo.h", if a class would be called "Foo"). Most of them inherit from `TNamed`, a generic class of ROOT which offers method for name and title bookkeeping. This is an important feature of the way ROOT-files are written (the name being the tag used to write or read any object and to handle them in memory). These classes belong to a specific namespace, in order to prevent any mixing between ROOT and Uranie (even though in practice, the name of the new classes implemented in Uranie is checked not to exist in ROOT for obvious safety and clarity reasons). Uranie defines many namespaces dedicated to the different module already introduced in Section I.1.1.

- *URANIE::DataServer* ; the namespace associated with the **DataServer** library
- *URANIE::Launcher* ; the namespace associated with the **Launcher** library
- *URANIE::Sampler* ; the namespace associated with the **Sampler** library
- *URANIE::Optimizer* ; the namespace associated with the **Optimizer** library
- *URANIE::Modeler* ; the namespace associated with the **Modeler** library
- *URANIE::Sensitivity* ; the namespace associated with the **Sensitivity** library
- *URANIE::Relauncher* ; the namespace associated with the **Relauncher** library
- *URANIE::MpiRelauncher* ; the namespace associated with the **Relauncher** library once used with MPI
- *URANIE::Calibration* ; the namespace associated with the **Calibration** library
- *URANIE::MetaModelOptim* ; the namespace associated with the **MetaModelOptim** library
- *URANIE::Reoptimizer* ; the namespace associated with the **Reoptimizer** library
- *URANIE::UncertModeler* ; the namespace associated with the **UncertModeler** library
- *URANIE::Reliability* ; the namespace associated with the **Reliability** library

You can use the full qualified name to access to Uranie classes or use the `using namespace` directives to use the short name.

Another way to use the short name is to use again the `rootlogon.C` macro. If done so, all macros could access Uranie classes via the short name implicitly. Here is an example of `rootlogon.C` file:

```
using namespace URANIE::DataServer;
using namespace URANIE::Launcher;
using namespace URANIE::Sampler;
using namespace URANIE::Optimizer;
using namespace URANIE::Modeler;
using namespace URANIE::UncertModeler;
using namespace URANIE::Sensitivity;
using namespace URANIE::Relauncher;
using namespace URANIE::Reoptimizer;
using namespace URANIE::Calibration;
using namespace URANIE::Reliability;
// using namespace URANIE::XMLProblem;
// using namespace URANIE::MpiRelauncher;

void rootlogon()
{
    gStyle->SetPalette(1);
    gStyle->SetOptDate(21);

    //General graphical style
    // Default colors
    int white = 0;
    int color = 30;

    //Legend
    gStyle->SetLegendBorderSize(0);
    gStyle->SetFillStyle(0);

    // Pads
    gStyle->SetPadColor(white);
    gStyle->SetTitleFillColor(white);
    gStyle->SetStatColor(white);

}

/* ==================== Hint ====================

   Might be practical to store this in a convenient place (for instance
   your home directory) and to create an alias to make sure that you use
   only one rootlogon file independently of where you are.

   example : alias root="root -l ${HOME}/rootlogon.C"

   Many style issue can be set once and for all here.

   Warnings :
    => The name of the main function (in between the void and the () part)
    has to be the same as the name of the file (without extension).
    => If you intend to change this file name and make it a hidden file (let's
    say  ${HOME}/.toto.C, the name of the main function would have to start with
    an underscore, so here it would be "void _toto()".
*/
```

The most generic solution to simply don't have to bother with namespaces while possibly tuning root graphical options to ones taste is to centralise this file and make it our own. By putting it somewhere central (for instance as a hidden file in your home folder, something like `~/.rootlogon.C`) it is possible to make an alias that could have two purposes:

C Shell Family (csh, tcsh, etc.)

```
alias root root -l ~/.rootlogon.C
```

Bourne Shell Family (sh, bash, etc.)

```
alias root="root -l ~/.rootlogon.C"
```

With this kind of alias, one can have two interesting by-products: on the one hand, there will be no splash screen anymore (resulting from the "-l" option) while on the other hand, all the Uranie namespaces will be loaded along with your own graphical preferences, if you have any.

I.2.5 Important modifications going from ROOT v5 to ROOT v6

This part summarises the important modifications that are brought when changing ROOT versions. This affects some of the Uranie macros and if you've used Uranie before version 4.X, this part can be useful to understand the modifications that will mainly affect the constructor of some Uranie's objects.

The main subject is the way to handle function. As an example, we will considerer the "very" simple function that with two inputs returns the sum of these two values (*c.f.* the `Addition` function below).

```
void Addition(double *x, double *y)
{
    y[0] = x[0] + x[1] ;
}
```

In C++, within the Uranie framework, there are three ways to use this function in a macro, that would be called `analysis.C`:

1. Having the function in a separated file `MyFunction.C` and load it through ROOT by calling

   ```
   gROOT->LoadMacro("MyFunction.C");
   ```

2. Having the function in a separated file `MyFunction.C` and include this file in the header of `analysis.C` as

   ```
   #include "MyFunction.C"
   ```

3. Having the function in the same file (`analysis.C`) before the main function.

This is common practice both for the 5 and 6 versions of ROOT.

From this, the way to handle this function is a little bit different going from one ROOT major-version (*i.e.* 5.X) to another one (*i.e.* 6.X).

ROOT 5.X: Disregarding the chosen option to read the function (1, 2, or 3 discussed above) the interpreter (CInt for these versions of ROOT) is associating to the pointer of the function `Addition` (which is a pointer of the form `(void *)(double *, double*)` a name, which it uses as a tag by CInt. The name is in this case "Addition". For the 3 configuration defined up there, one can use the name and/or the pointer to access the function.

ROOT 6.X: On the other hand, for these versions of ROOT, the interpreter is, as stated previously, a runtime compiler, so it behaves very much like expected from properly C++ compiled code. When calling the function thanks to its pointer (the `(void *)(double *, double*)` object) no information on a name is accessible. Compared to ROOT 5, there is no pointer to the function if it is loaded through `LoadMacro` (which is perfectly logic from a C++ point of view). The second and third access method can, on the other hand, use both the pointer and the name.

This obliges us to change several constructor that had been (over ?) simplified: in ROOT 5 versions, the compulsory information were often just a pointer to the `TDataServer` object and either the name of the function or a pointer to it. From this, if nothing else was specified, all current inputs in the `TDataServer` were used as inputs (this behaviour could be changed usually providing at third argument) and the output variables were named using the tag taken from CInt (this behaviour could be changed as well usually providing a fourth argument). Having no tag anymore from CInt when running ROOT 6, the (usually) optional third and fourth arguments become now compulsory.

I.2.6 References

The ROOT environment [**?**] has already been under development for a long time now, it's documentation is very well advanced and is organised with different level of precision. One can indeed find many useful information in:

- The ROOT user guide (https://root.cern.ch/root/htmldoc/guides/users-guide/ROOTUsersGuideA4.pdf): useful and relatively complete gathering of information. It might be helpful to keep it locally as a reference.
- The ROOT Class index (https://root.cern.ch/root/html/ClassIndex.html): Very complete description of all the classes and their methods. One can spent hour passing through these pages to discover what's doable using the various objects. It is even possible to mask/show inherited method for sake of simplicity.

There are also many examples and macros that show how to handle the objects provided:

- The Howto website: https://root.cern.ch/howtos
- All the macro contained in the tutorial folder, installed once ROOT has been installed (check in the $ROOTSYS/tutorials folder).

I.3 The Python Interface

Accessing Uranie tools is also possible using the Python language.

The **PyROOT** tool allows to access to ROOT classes from a Python command line or script. Uranie can benefit from this tool as well. All the use-case macros provided in the previous version of this user manual (*c.f.* Chapter XIV) are now available in Python in the python version of this manual.

I.3.1 Python version: greater than 3.8

From Uranie version 4.9, only Python greater than 3.8 can be used due to the use of ROOT version v6.32 (Python 2 is deprecated). Two words of caution about this:

- The macros provided in the python version of this manual have indeed been tested with Python 3 (upper than 3.8).
- Note, historically, from Uranie version 4.5 to v4.8, both Python 2 and 3 were used at the same time if it is installed again a ROOT version from v6.20.00 to v6.28.06. In order to get this compatibility, it means that the printing format has to be homogenised and this implies that from Uranie version 4.2, these macros might not be working anymore for Python version below or equal to 2.6.

Possible python configuration on a Linux OS

```
ls -l `which python`  && ls -l `which python3`
```

```
lrwxrwxrwx. 1 root root 9  8 fÃ©vr.  2023 /usr/bin/python -> ./python3
lrwxrwxrwx. 1 root root 10  8 fÃ©vr.  2023 /usr/bin/python3 -> python3.11
```

Today's operating systems usually embed both versions of python and many of these still use the version 2 as a reference (see the block above with both python versions). Given the fact that one can have as many different python versions as possible, it should be possible to specify which one is of interest for the ongoing installation. The following lines should be used to specify the python version to be used, providing that the "dev" packages are indeed installed for this version:

```
cmake ${PATH_TO_ROOT_SOURCES} -DPYTHON_EXECUTABLE=${PATH_TO_PYTHON_BIN} - ↩
    DPYTHON_INCLUDE_DIR=${PATH_TO_PYTHON_INC}  -DPYTHON_LIBRARY=${PATH_TO_PYTHON_LIB} ...
```

These three specific python flags should be used both for ROOT and Uranie in a coherent way in order to install both platforms with a chosen Python's version. To get the recommended *cmake* line both for ROOT and Uranie see the README provided with the Uranie-sources.

I.3.2 Environment variables

In order to access ROOT and Uranie from Python, we need to ensure that some environment variables are properly set:

- *$PYTHONPATH* must contain the *$ROOTSYS/lib* directory, where *$ROOTSYS* is ROOT's installation directory.
- *$LD_LIBRARY_PATH* must contain the *$ROOTSYS/lib* directory.

I.3.3 Using PyROOT

When the environment variables for Uranie and PyROOT are properly set, we can access to the classes as follows:

```
# Load the ROOT module
import ROOT

# Create a new data server object
tds=ROOT.URANIE.DataServer.TDataServer("myTDS","DataServer for python example")

# Add an attribute to the data server
tds.addAttribute( ROOT.URANIE.DataServer.TNormalDistribution("x", 0.0, 1.0) )

# Create a sampler object
sampler = ROOT.URANIE.Sampler.TSampling(tds, "lhs", 1000)

# Generate data
sampler.generateSample()

# Display the histogram of attribute x
tds.draw("x")
```

This code should produce a graphic as the one displayed in Figure I.2.

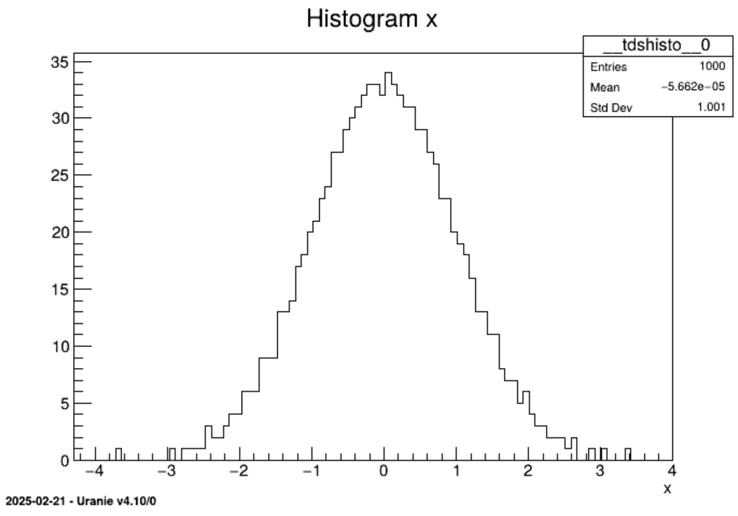


Figure I.2: Histogram produced using PyROOT

The instructions above can be executed either through the Python command line, or be written in a file (*myscript.py* in the example below) and run using the command:

```
python -i myscript.py
```

The *-i* option allows to stay in the Python environment at the end of the execution. This prevents the produced image to be automatically closed.

I.3.4 The PyURANIE interface

In the previous example, we can see that the access to Uranie's classes is somewhat tedious. In order to ease the process, a set of specific modules have been created. We call it the *PyURANIE interface*.

It is then possible to re-write the previous example using these specific modules:

```python
# Load the URANIE module
from ROOT import URANIE

# Load the DataServer and Sampler modules
from URANIE import DataServer, Sampler

# Create a new data server object
tds = DataServer.TDataServer("myTDS","DataServer for the python example")

# Add an attribute to the data server
tds.addAttribute( DataServer.TNormalDistribution("x", 0.0, 1.0) )

# Create a sampler object
sampler = Sampler.TSampling(tds, "lhs", 1000)

# Generate data
sampler.generateSample()

# Display the histogram of attribute x
tds.draw("x")
```

The access to ROOT classes is provided through the ROOT module.

The PyURANIE interface also allows to use the command:

```
from URANIE.DataServer import *
```

This command loads all the classes of the DataServer module in Python and makes them directly accessible. It is similar to the C++ command *using namespace URANIE::DataServer*. However, in Python, using this command is not recommended. It can create name conflicts and use a large amount of memory.

Finally the equivalent of the `rootlogon.C` has been written for python, and is called `rootlogon.py`. It is composed of two parts, as for the one in C++, the second one being the exact equivalent. The first one, on the other hand, is a bit different and this difference arises from the way the language are dealing with loading modules. By doing

```
from rootlogon import DataServer
```

one can, for instance, directly creates a `TDataServer` by doing

```
toto=DataServer.TDataServer()
```

The example of rootlogon file for python is the following:

```
import ROOT

# Create shortcuts if uranie exists
urasys = ROOT.TString(ROOT.gSystem.Getenv("URANIESYS"))
if not urasys.EqualTo(""):
    from ROOT.URANIE import DataServer as DataServer
    from ROOT.URANIE import Sampler as Sampler
    from ROOT.URANIE import Launcher as Launcher
    from ROOT.URANIE import Relauncher as Relauncher
    from ROOT.URANIE import Reoptimizer as Reoptimizer
    from ROOT.URANIE import Sensitivity as Sensitivity
    from ROOT.URANIE import Optimizer as Optimizer
    from ROOT.URANIE import Modeler as Modeler
    from ROOT.URANIE import Calibration as Calibration
    from ROOT.URANIE import UncertModeler as UncertModeler
    from ROOT.URANIE import Reliability as Reliability
    from ROOT.URANIE import XMLProblem as XMLProblem
    from ROOT.URANIE import MpiRelauncher as MpiRelauncher
    pass

# General graphical style
white = 0

# PlotStyle
ROOT.gStyle.SetPalette(1)
ROOT.gStyle.SetOptDate(21)

# Legend
ROOT.gStyle.SetLegendBorderSize(0)
ROOT.gStyle.SetFillStyle(0)

# Pads
ROOT.gStyle.SetPadColor(white)
ROOT.gStyle.SetTitleFillColor(white)
ROOT.gStyle.SetStatColor(white)

#   ====================  Hint ====================
#
#    Might be practical to store this in a convenient place (for instance
#    the ".python" folder in your home directory) or any other place where
#    your $PYTHONPATH is pointing.
#
#    example : export PYTHONPATH=$PYTHONPATH:${HOME}/.mypython/
#
#    It should then be called as "from rootlogon import " + the list of module
#    This would replace the shortcuts created and import done in the rest of
#    the scripts
#
#    Many style issue can be set once and for all here.
#    toto=DataServer.TDataServer()
#
```

I.3.5 References

The PyROOT environment has a few specificities which it is preferable to be aware of (the rest being discussed already in Section I.2.6). The following websites help to learn about them:

- Presentation on the PyROOT website: http://root.cern.ch/drupal/content/pyroot
- The PyROOT Manual: http://wlav.web.cern.ch/wlav/pyroot/index.html
- The ROOT users forum: http://root.cern.ch/phpBB3/index.php

Finally, reference [1] introduces (in French) some of these problems and shows examples on how to use Uranie with PyROOT.

Chapter II

The DataServer module

II.1 Introduction

The DataServer module is the spine of the Uranie platform as it is where the data are stored, the attributes (name given to the variable in Uranie) are gathered and important basic mathematical operations are performed. Objects and methods will indeed need a `TDataServer` to retrieve, process and transmit the results of their own operations.

Since it is used by all other technical libraries (Sampler, Launcher, ...), this library is the core library of Uranie (as shown in Figure I.1). The already discussed `TDataServer` contains two main objects (has shown in Figure II.1):

- the **header**: represented in Uranie by the object `TDataSpecification`. Different information related to the variables (called in "attributes" in Uranie language and represented by `TAttribute` objects) are specified in Section II.2.
- the **data matrix**: represented by the object `TDSNtupleD` (class derived from the ROOT class `TTree`). For an advanced use of this object, the reader can refer to chapter XII "Trees ROOT" in ROOT's user manual.

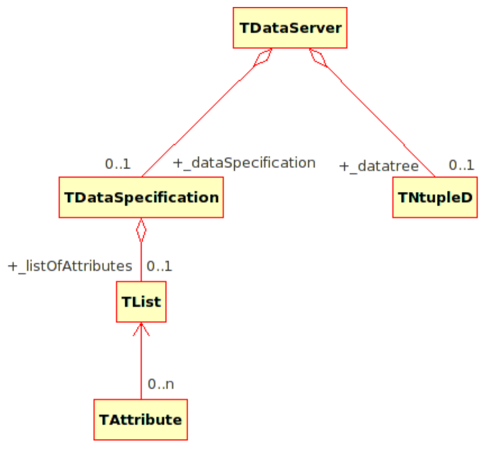


Figure II.1: Diagram of the class `TDataServer`

The chapter presentation will be articulated as follows. First of all, the way variable are handled will be introduced in Section II.2. This is a needed step, as all `TDataServer` have to be filled with attributes for being able to move along. The creation of a `TDataServer` (with many construction, from *ASCII* to ROOT files) is discussed in Section II.3 before discussing the basic statistical treatment (in Section II.4) and the dedicated visualisation tool (in Section II.5).

Tip

An important point to know for the use of Uranie: all options are not case sensitive. Actually, the treatment of these options is taken anyway as lowercase. The following instructions are equivalent:

```
tdsGeyser->draw("x1","","Nclass=Sturges");
tdsGeyser->draw("x2","","nclass=Sturges");
tdsGeyser->draw("x2","","nclass=sturges");
tdsGeyser->draw("x2","","NCLASS=STURGES");
```

II.2 The TAttribute class

The `TAttribute` class (as its inherited classes, some of which are discussed here as well), is also a crucial part of any analysis performed through Uranie. It describes any variable (input, output, or internal variable such as iterator) that are passed to the other modules. It does not really contain the data, but it has the statistical information (in case methods such as `computeStatistic` or `computeQuantile` have been called, see Section II.4)

II.2.1 Nature of the attribute

Unlike previous Uranie-version where all attribute were double-precision float values, the implementation done from v3.10.0 allows to handle two other types of attribute: string and vector. There are several ways to define the new attribute nature, following the chosen construction process, but all these methods will affect the enumerator `URANIE::DataServer`
whose value can be:

- `kDefault`: the default one which is equal to `kReal`
- `kString`: in the case of text input (assuming that this text is no split into more than one word, unless extra-cautions are taken)
- `kVector`: for vectors of double-precision values. Even though the number of elements within this vector can change from one pattern to the other, many useful methods discussed in the following sections and chapters will required (in order to make sense) to have a constant number of elements (at least for a sub-selection of patterns). This is particularly true for all mathematical methods and sensitivity analysis. The code should complain if this requirement is needed and not fulfilled.

This new implementation is bringing changes in the way some information are handled, as for the attribute is concerned, but all the resulting modification have been checked to be backward compatible. **Default settings are made assuming that attributes are double-precision ones, unless specified otherwise (to be sure that all previous script will work).** The corresponding modifications are discussed throughout this documentation.

II.2.2 List of variable information

We will present in this section the list of information contained in a `TAttribute` of Uranie.

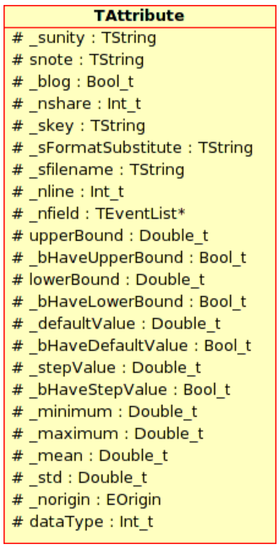


Figure II.2: Attributes of TAttribute class

Name: Variable name It should be a *short* name as this information is needed to use this variable (mathematical expressions, graphics, scan, ...).

Title: Variable title. This information is only needed for graphical display.

Unit: Variable units. This information is only needed for graphical display.

Note: Variable note. description of the variable, this is not currently used.

Min, Max, Mean and Std: Minimum, maximum, averaged and standard deviation values. These information are now vectors and their usage is discussed in Section II.4.3

vquantile: Vector of map containing value of the quantile computed using the key in argument. These information are now stored in the attribute itself their usage is discussed in Section II.4.4

defaultValue: Default value. This default value will be considered either by the code launcher or during a parameter optimisation. In the case of a code launcher, this means either that the code failed to proceed or that the code did not return the value.

At this level, there is no notion of **random variable**. Attributes are variables with a name, a label, a unit, a variation domain (bounded or belonging to R). Despite the large amount of possible combinations to instantiate a variable (name, name+label, name+boundary, name+label+boundary), only a small number of constructors are implemented. Some methods like **setTitle**, **setFileKey**, **setUnity** allow to precise the missing information.

The four constructors currently implemented are the following:

- Name: since this constructor only knows the name of the variable, the 3 piece of information title, label and key are strictly identical. This variable is not bounded. An example of use, already seen before, is:

  ```
  TAttribute *px = new TAttribute("x");
  ```

  A random variable **"x"** (where px denotes a pointer to x) exists and it has its label also equal to x. Thus, if this variable is visualised on a graph, the default label will also be its title i.e **"x"**.

- Name + title: constructor defined from the name and the title of the variable

  ```
  TAttribute *psdp = new TAttribute("sdp", "#sigma_{#Delta P}");
  psdp->setUnity("M^{2}");
  ```

  A pointer **psdp** to a variable **"sdp"** is available with title being *#sigma_{#Delta P}*. The command **setUnity()** precises the unit. In this case, by default, the field *key* is identical to the field *name*. We will use the ability given by ROOT to write LaTeX expressions in graphics to improve graphics rendering without weighing down the manipulation of variables: as a matter of fact, we can plot the histogram of the variable *sdp* by:

  ```
  tdsGeyser->addAttribute("newx2","x2","#sigma_{#Delta P}","M^{2}");
  tdsGeyser->draw("newx2");
  ```

  The result of this piece of code is shown in Figure II.3.

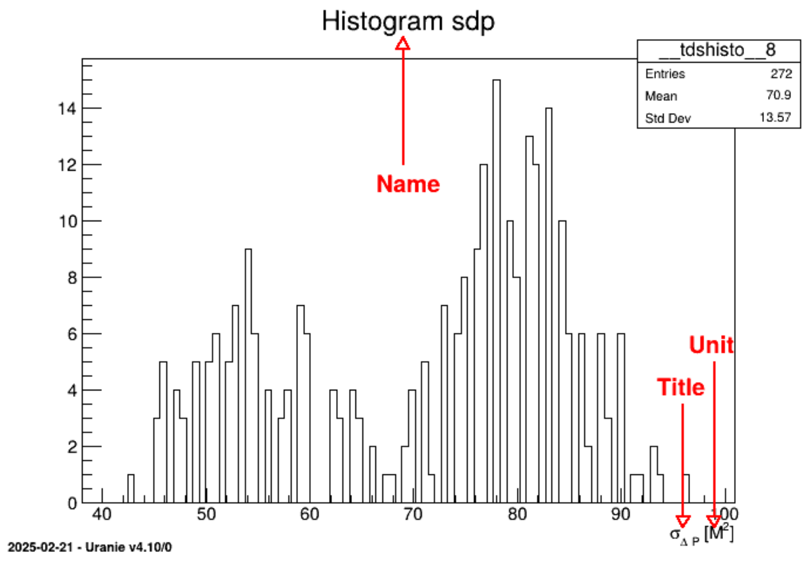


Figure II.3: Graph of the variable *sdp*

- Name + variation boundary: constructor defined by the name of the variable and the lower and upper boundaries. The two other pieces of information, label and key remain equal to the title. An example of use is

  ```
  TAttribute *x = new TAttribute("x", -2.0, 4.0);
  ```

- Name + EType: constructor defined by the name of the variable and nature of the corresponding attribute . An example of use is

  ```
  TAttribute *xvec = new TAttribute("x", URANIE::DataServer::TAttribute::kVector);
  ```

Setter methods allow to fill the other fields (title, key, etc) generally by calling **set** and the name of the information to be modified (a restricted list of available methods being given below). For instance, the plot "x2:x1" of `TDataServer` data *tdsGeyser* (whose data file `geyser.dat` can be found in the Uranie-macros folder) can be considered again and we can replace the fields *title* and *unit* with new values by using **LaTeX** instructions. For instance, let us consider once again the graph of `TDataServer` data tdsGeyser:

```
TAttribute *px1 = tdsGeyser->getAttribute("x1");
px1->setTitle("#Delta P^{#sigma}"); // Change the title
px1->setUnity("#frac{mm^{2}}{s}"); // Change the unit
tdsGeyser->Draw("x2:x1"); // Draw the plot
```

The first line consists in retrieving the attribute pointer *x1*, while the others are self explanatory. This results in a new graph (scatterplot) of *x2* versus *x1* for the `TDataServer` constructed from the *geyser* file with updated field title and unit values, shown in Figure II.4.

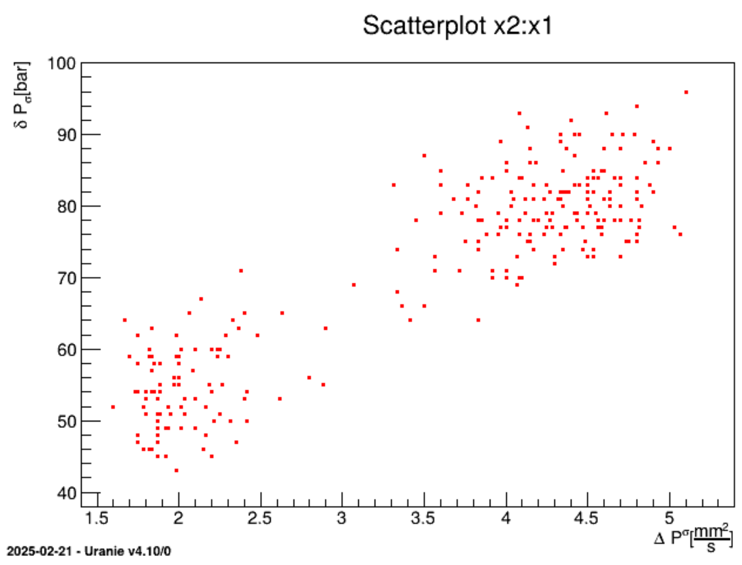


Figure II.4: Scatterplot *x2 versus x1* for the geyser data with modification of fields *title* and *unit*.

Most of the information can be modified by "setter" methods. Here is a short list of the most relevant one starting with simple and already discussed attribute properties:

- **setTitle**(TString *str*): assigns the character string *str* passed as argument to the field **title**;
- **setUnity**(TString *str*): assigns the character string *str* passed as argument to the field **unity**;
- **setNote**(TString *str*): assigns the character string *str* passed as argument to the field **note**;
- **setUpperBound/setLowerBound/setDefaultValue**(double *val*): assigns or changes (if it already existed) respectively the upper, lower or default value for this attribute.
- **setDataType**(EType *thetype* or TString *str*): changes the nature of the attribute given the enumerator value or a character chain (case insensitive). In the latter case, the enumerator is set to:
  - kReal: *str*= "double" or "real" or "d";
  - kString: *str*= "string" or "s"
  - kVector: *str*= "vector" or "v"

Given the new nature of attributes (meaning vectors and strings) a more generic method has been created to put default values to all type. The generic methods takes only one argument, a string containing values whatever the type. In cas of doubt, dedicated methods have also been created, with dedicated prototypes:

- kReal: both `setDefault(TString value)` and `Bool_t setDefaultValue(Double_t val)` can be used as shown below

```
TAttribute *real = new TAttribute("real");
double real_value=1.23456789;
real->setDefaultValue(real_value); // Default with double value
real->setDefault("1.23456789"); // Default with generic method
```

- kVector: both `setDefault(TString value)` and `setDefaultVectorvector<double> &vec)` can be used as shown below

```
TAttribute *vector = new TAttribute("vector",TAttribute::kVector);
std::vector<double> v_value={1.2,2.3,3.4};
vector->setDefaultVector(v_value); // Default with double value
vector->setDefault("1.2,2.3,3.4"); // Default with generic method
```

- kString: both `setDefault(TString value)` and `setDefaultString(TString val)` can be used as shown below

```
TAttribute *string = new TAttribute("string",TAttribute::kString);
TString str_value="chocolat";
string->setDefaultString(str_value); // Default with double value
string->setDefault(str_value); // Default with generic method
```

There are also important setters, used to connect attributes to *ASCII* files. Most of the time, *ASCII* files are indeed used to communicate with an external code and Uranie must know in this case where to find the useful information for the corresponding attributes (either to write a new value that would be used as input to perform a calculation or to read the output of another computation). This is more carefully detailed in Section IV.3.1.

- **setFileKey**(TString *sfile*, TString *skey*,TString *sformatToSubstitute*, TAttributeFileKey::EFileType *sFileType*): allows to specify for an attribute a file **sfile**, a key linked to this file **skey**, a writing format of the value of this key in the previous file **sformatToSubstitute**, and also the type of the file **sFileType**. This is heavily discussed in Chapter IV.

II.2.3 Examples of use of the class TAttribute

The following instruction defines a pointer **px** to an unbounded variable **x**.

```
TAttribute *px = new TAttribute("x");
```

The following instruction defines a pointer **px** to a variable **x** bounded between 0. and 1.

```
TAttribute *px = new TAttribute("x", 0., 1.);
```

The following instruction defines a pointer **px** to a variable **x** bounded between -2. and 4. with $\Delta P_e^{F_{iso}}$ as label:

```
TAttribute *px = new TAttribute("x", -2.0, 4.0);
px->setTitle("#Delta P_{e}^{F_{iso}}");
```

The following instruction defines a pointer **px** to a variable **x** that describes string

```
TAttribute *px = new TAttribute("x", URANIE::DataServer::TAttribute::kString);
```

II.2.4 Adding TAttribute when data are already available

It is possible to add new attribute in a given `TDataServer` object that would contain data using two different methods. The first one rely on the already existing data to create a new variable by simply writting the equation, which internally is calling a `TAttributeFormula` object. The following piece of code shows how to create a third variable from the `geyser.dat` file, simply as an equation from the existing variables:

```
TDataServer *tds = new TDataServer("foo","pouet");
tds->fileDataRead("geyser.dat");
// Adding a new attribute
TAttribute *x3 = new TAttribute("x3","0.5*x2+sin(x1)");
```

This method can deal with double and vector-based attributes (of course no equation can be estimate when one of the input variable in the formula is a string one, so this method will crash).

Another way recently introduced is to add an attribute from an array of double (or it's equivalent in `python`, meaning a `numpy.array`) by calling the method `addAttributeUsingData`. The idea is to be able to add information that would have been processed by methods aside from the Uranie ones. The signature of the function is the name of the new attribute as first argumet, the second one is an array of double and the third one is the size of the array. There are two ways, recommended, that uses object with built-in method that provide the size (to prevent from mis-typing problem between the array and its size). Obviously, the size of the array must be equal to the number of patterns in the existing `TDataServer` object. Here is an example using the `myData.dat`, in C++:

```
TDataServer *tds = new TDataServer("foo","tru");
tds->fileDataRead("myData.dat");
// Defining a vector with 11 elements
vector<double> x2 = {-10,-8,-6,-4,-2,0,2,4,6,8,10};
// Call the method using the address of first element and the size of it
tds->addAttributeUsingData("x2", &x2[0], x2.size());
```

This method should only be used to create double-based attributes (as the size of the array would be chaotic if it were to be a vector of varying size). Obviously, no string-based attribute can be constructed like this.

Warning The method `addAttributeUsingData` should only be called either when no data AND no attribute are stored in the dataserver or when there are data and the new array of double provided has the same size (number of patterns) as the data already available within the dataserver.

II.2.5 Introducing the `TStochasticAttribute` classes

The `TStochasticAttribute` is the parent class to all attributes which values can be generated by a `TSampler` (as discussed in Section III.2). All child objects are random variables, following a specific law, that depends on a small number of parameters.

As from version 4.8 of the Uranie platform it is possible to combine different probability law, as a sum of weighted contributions, in order to create a new law. This approach, which is further discussed and illustrated in Section II.2.5.19, leads to a new probability density function that would look like

$$f(x) = \sum_{j=1}^{N} \omega_j f_j(x) \text{ where } \forall j \in [1,N],\ \omega_j \in \mathbb{R}^+.$$

These distributions can be used to model the behaviour of variables, depending on chosen hypothesis, probability density function being used as a reference more oftenly by physicist, whereas statistical experts will generally use the cumulative distribution function [3].

Table II.1 gathers the list of implemented statistical laws, along with its class name in Uranie and the list of parameters used to define them. For every possible law, a piece of code is provided to show how to draw a simple PDF, along with a figure that displays the PDF, CDF and inverse CDF[1] for different sets of parameters (the equation of the corresponding PDF is reminded as well on every figure). The inverse CDF is basically the CDF whose x and y-axis are inverted (it is convenient to keep in mind what it looks like, as it will be used to produce design-of-experiments, later-on). For all these laws, the parameters can be set at the constructor (as shown in the previous example block) but, if this has not been done it is possible to change their value using the `setParameters` method.

To define a random variable, the corresponding constructor must be used. The arguments of these constructors are first, the name of the variable and second, the parameters of the law. For example:

[1] for a definition of PDF (*probability density function*), CDF (*cumulative density function*) and inverse CDF, please look at [30]

Law	Class Uranie	Parameter 1	Parameter 2	Parameter 3	Parameter 4
Uniform	TUniformDistribution	Min	Max		
Log-Uniform	TLogUniformDistribution	Min	Max		
Triangular	TTriangularDistribution	Min	Max	Mode	
Log-Triangular	TLogTriangularDistribution	Min	Max	Mode	
Normal (Gauss)	TNormalDistribution	Mean (μ)	Sigma (σ)		
Log-Normal	TLogNormalDistribution	Mean (M)	Error factor (E_f)	Min	
Trapezium	TTrapeziumDistribution	Min	Max	Low	Up
UniformByParts	TUniformByPartsDistribution	Min	Max	Median	
Exponential	TExponentialDistribution	Rate (λ)	Min		
Cauchy	TCauchyDistribution	Scale (γ)	Median		
GumbelMax	TGumbelMaxDistribution	Mode (μ)	Scale (β)		
Weibull	TWeibullDistribution	Scale (λ)	Shape (k)	Min	
Beta	TBetaDistribution	alpha (α)	beta (β)	Min	Max
GenPareto	TGenParetoDistribution	Location (μ)	Scale (σ)	Shape (ξ)	
Gamma	TGammaDistribution	Shape (α)	Scale (β)	Location (ξ)	
InvGamma	TInvGammaDistribution	Shape (α)	Scale (β)	Location (ξ)	
Student	TStudentDistribution	DoF (k)			
GeneralizedNormal	TGeneralizedNormalDistribution	Location (μ)	Scale (α)	Shape (β)	

Table II.1: List of Uranie classes representing the probability laws

```
//Uniform law
TUniformDistribution *pxu = new TUniformDistribution("x1", -1.0 , 1.0); ❶[1]
// Gaussian Law
TNormalDistribution *pxn = new TNormalDistribution("x2", -1.0 , 1.0); ❷[2]
```

❶ Allocation of a pointer *pxu* to a random uniform variable *x1* in interval [-1.0, 1.0].

❷ Allocation of a pointer *pxn* to a random normal variable *x2* with mean value μ=-1.0 and standard deviation σ=1.0.

These distributions can be used to model the behaviour of inputs, the choice being generally based on the way the PDF looks like. For every distributions implemented in Uranie examples of PDF, CDF and inverse CDF are show from Figure II.5 until Figure II.28. Here is a brief description of the probability density functions and their parameters.

II.2.5.1 Uniform Law

The Uniform law is defined between a minimum and a maximum, as

$$f(x) = \frac{1}{(x_{\max} - x_{\min})} \mathbb{I}_{[x_{\min},x_{\max}]}(x)$$

Uranie code to simulate an uniform random variable is:

```
TDataServer *tds = new TDataServer("tdssampler", "Sampler Uranie demo");
tds->addAttribute( new TUniformDistribution("u", -2., 3.));
```

```
TSampling *fsamp = new TSampling(tds, "lhs", 300);
fsamp->generateSample(); // Create a representative sample

tds->Draw("u");
```

Figure II.5 shows the PDF, CDF and inverse CDF generated for a given set of parameters.

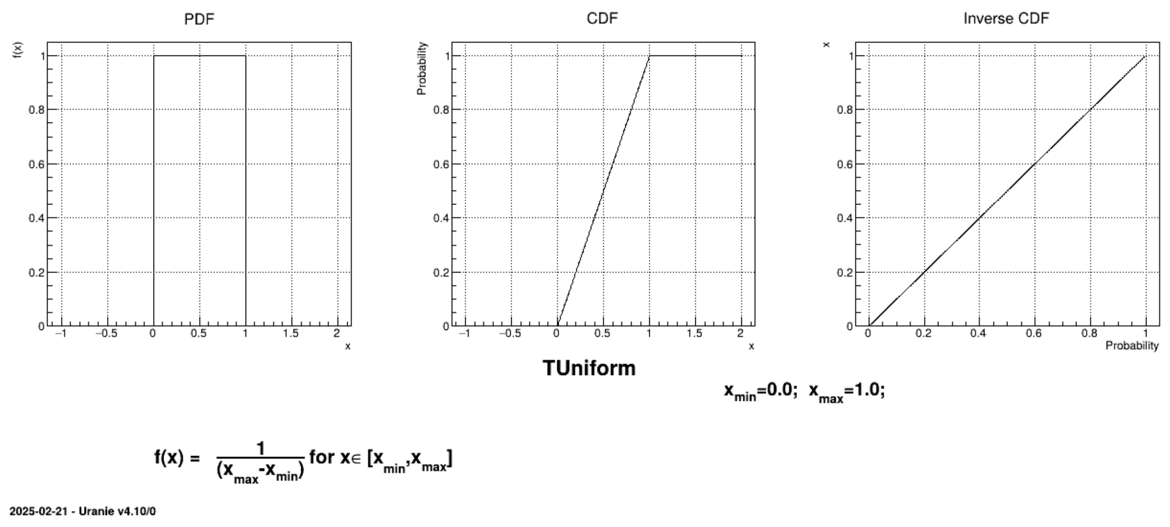


Figure II.5: Example of PDF, CDF and inverse CDF for Uniform distribution.

II.2.5.2 Log Uniform Law

The LogUniform law is well adapted for variations of high amplitudes. If a random variable x follows a LogUniform distribution, the random variable $\ln(x)$ follows a Uniform distribution, so

$$f(x) = \frac{1}{(x \times \ln(x_{\max}/x_{\min}))} \mathbb{I}_{[x_{\min},x_{\max}]}(x)$$

Uranie code to simulate a LogUniform random variable is:

```
TDataServer *tds = new TDataServer("tdssampler", "Sampler Uranie demo");
tds->addAttribute( new TLogUniformDistribution("lu", .001, 10.));

TSampling *fsamp = new TSampling(tds, "lhs", 300);
fsamp->generateSample(); // Create a representative sample

tds->Draw("lu");
tds->Draw("log(lu)"); // Check that ln(x) follows a uniform law
```

Figure II.6 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

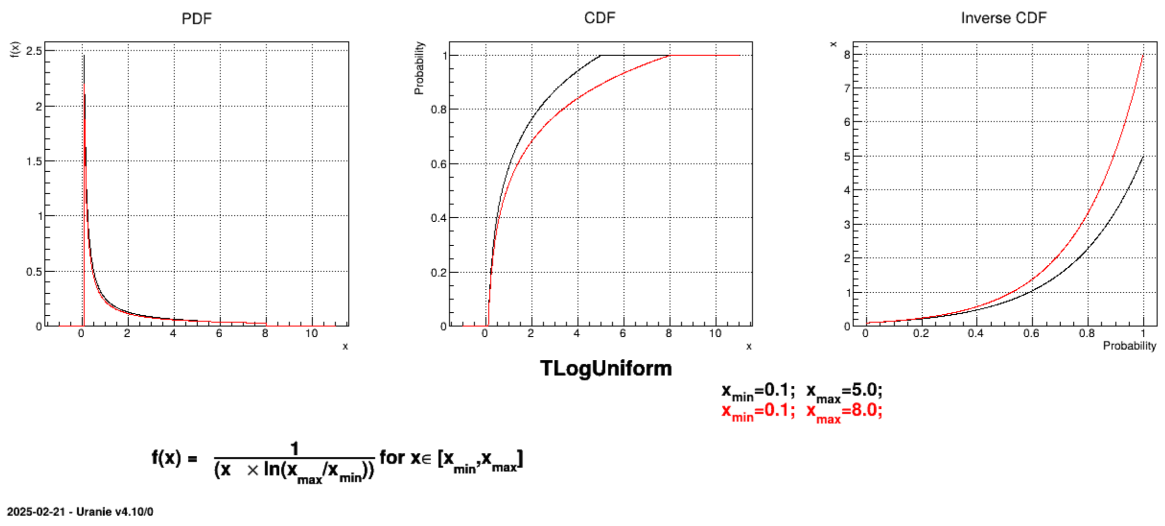


Figure II.6: Example of PDF, CDF and inverse CDF for LogUniform distributions.

II.2.5.3 Triangular law

This law describes a triangle with a base between a minimum and a maximum and a highest density at a certain point x_{mode}, so

$$f(x) = \frac{2\times(x-x_{\text{min}})}{(x_{\text{max}}-x_{\text{min}})\times(x_{\text{mode}}-x_{\text{min}})}\,\mathbb{I}_{[x_{\text{min}},x_{\text{mode}}]}(x) \quad\text{and}\quad f(x) = \frac{2\times(x_{\text{max}}-x)}{(x_{\text{max}}-x_{\text{min}})\times(x_{\text{max}}-x_{\text{mode}})}\,\mathbb{I}_{[x_{\text{mode}},x_{\text{max}}]}(x)$$

Uranie code to simulate a triangular random variable is:

```
TDataServer *tds = new TDataServer("tdssampler", "Sampler Uranie demo");
tds->addAttribute( new TTriangularDistribution("t", 5.0, 8., 6.0));

TSampling *fsamp = new TSampling(tds, "lhs", 300);
fsamp->generateSample(); // Create a representative sample

tds->Draw("t");
```

Figure II.7 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

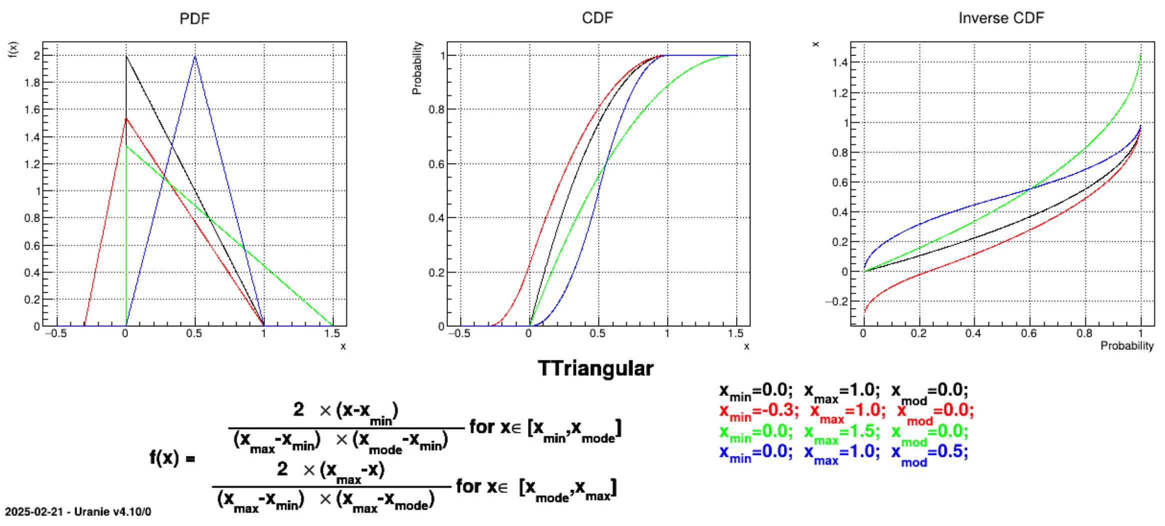


Figure II.7: Example of PDF, CDF and inverse CDF for Triangular distributions.

II.2.5.4 LogTriangular law

If a random variable x follows a LogTriangular distribution, the random variable $\ln(x)$ follows a Triangular distribution, so

$$f(x) = \frac{2 \times \ln(x/x_{\min})}{x \times \ln(x_{\max}/x_{\min}) \times \ln(x_{\text{mode}}/x_{\min})} \mathbb{I}_{[x_{\min}, x_{\text{mode}}]}(x)$$

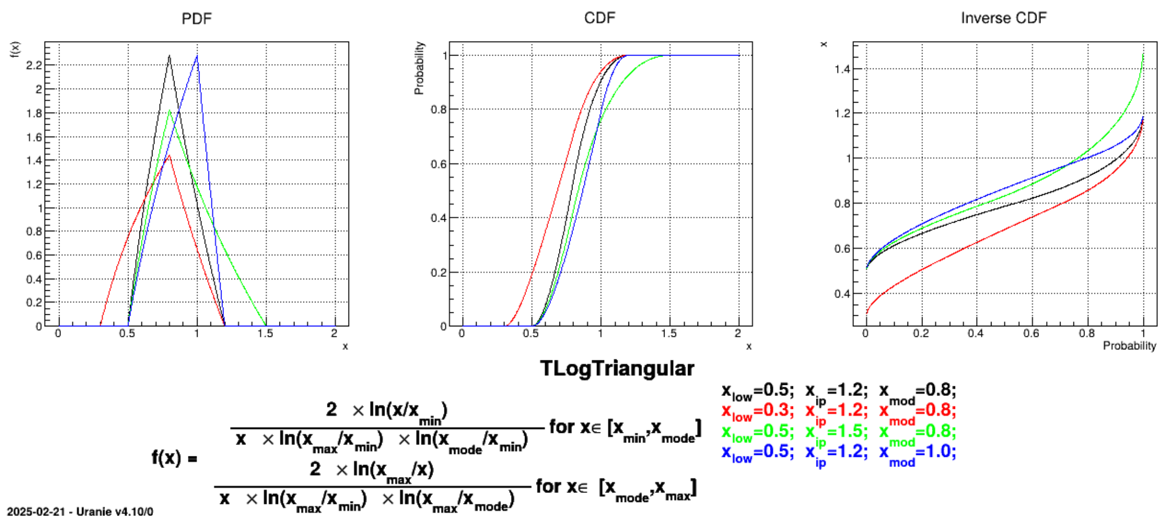
and

$$f(x) = \frac{2 \times \ln(x_{\max}/x)}{x \times \ln(x_{\max}/x_{\min}) \times \ln(x_{\max}/x_{\text{mode}})} \mathbb{I}_{[x_{\text{mode}}, x_{\max}]}(x)$$

Uranie code to simulate a LogTriangular random variable is:

```
TDataServer *tds = new TDataServer("tdssampler", "Sampler Uranie demo");
tds->addAttribute( new TLogTriangularDistribution("lt", .001, 10., 2.5));

TSampling *fsamp = new TSampling(tds, "lhs", 300);
fsamp->generateSample(); // Create a representative sample

tds->Draw("lt");
tds->Draw("log(lt)"); // Check that ln(lt) follows a triangular law
```

Figure II.8 shows the PDF, CDF and inverse CDF generated for different sets of parameters.



$$f(x) = \begin{cases} \frac{2 \times \ln(x/x_{min})}{x \times \ln(x_{max}/x_{min}) \times \ln(x_{mode}/x_{min})} & \text{for } x \in [x_{min}, x_{mode}] \\ \frac{2 \times \ln(x_{max}/x)}{x \times \ln(x_{max}/x_{min}) \times \ln(x_{max}/x_{mode})} & \text{for } x \in [x_{mode}, x_{max}] \end{cases}$$

Figure II.8: Example of PDF, CDF and inverse CDF for Logtriangular distributions.

II.2.5.5 Normal law

A normal law is defined with a mean μ and a standard deviation σ, as

$$f(x) = e^{\frac{-(x-\mu)^2}{2\sigma^2}} \times \frac{1}{\sqrt{2\pi\sigma^2}}$$

Uranie code to simulate a normal random variable is:

```
TDataServer *tds = new TDataServer("tdssampler", "Sampler Uranie demo");
tds->addAttribute( new TNormalDistribution("n", 0.0, 1.0));

TSampling *fsamp = new TSampling(tds, "lhs", 300);
fsamp->generateSample(); // Create a representative sample

tds->Draw("n");
```

Figure II.9 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

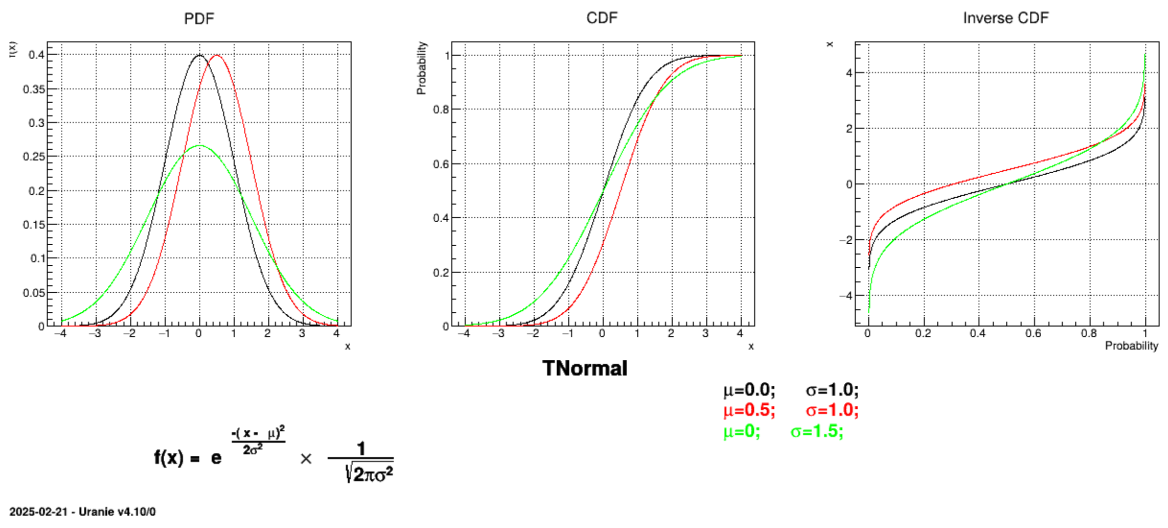


Figure II.9: Example of PDF, CDF and inverse CDF for Normal distributions.

Is it also possible to set boundaries to the infinite span of this distribution to create a truncated normal law. This can be done by calling the following method:

```
tds->getAttribute("n")->setBounds(-1.4,2.0); //truncate the law
```

The resulting PDF, CDF and inverse CDF, with and without truncation, can be seen, in this case, in Figure II.10 for a given set of parameters and various boundaries.

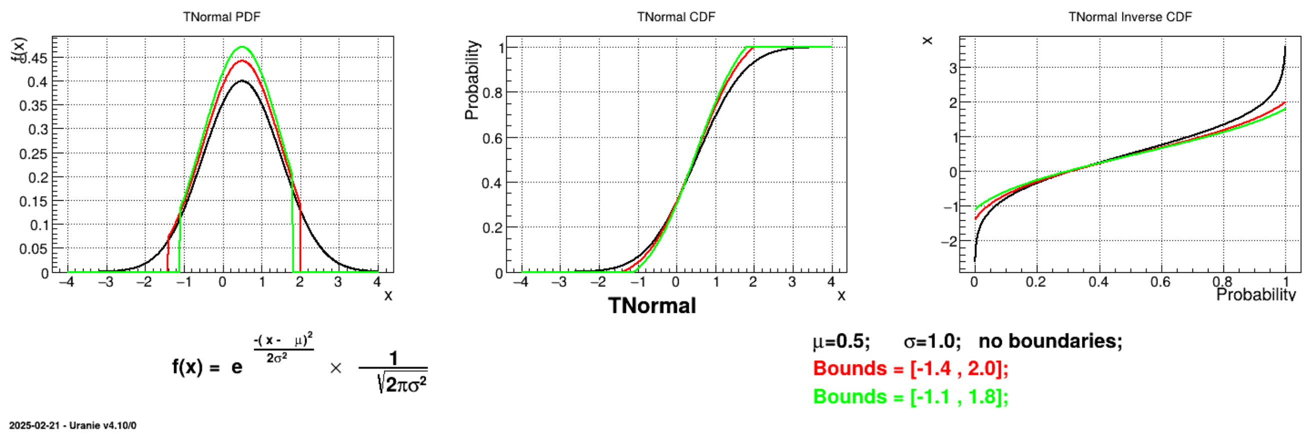


Figure II.10: Example of PDF, CDF and inverse CDF for a Normal truncated distribution.

II.2.5.6 LogNormal law

If a random variable x follows a LogNormal distribution, the random variable $\ln(x)$ follows a Normal distribution (whose parameters are μ and σ), so

$$f(x) = \frac{1}{(x-x_{\min})\sigma\sqrt{2\pi}} \times e^{\frac{-(\ln(x-x_{\min})-\mu)^2}{2\sigma^2}} \mathbb{I}_{[x_{\min},+\infty[}(x)$$

In Uranie, it is parametrised by default using M, the mean of the distribution, E_f, the Error factor that represents the ration of the 95% quantile and the median ($E_f = q_{0.95}/q_{0.50}$) and the minimum $x_{\min}$. One can go from one parametrisation to the other following those simple relations

$$\begin{array}{rcl} M = e^{\mu+\sigma^2/2} + x_{\min} & \Leftrightarrow & \mu = \ln(M - x_{\min}) - \sigma^2/2 \\ E_F = e^{1.645\times\sigma} & \Leftrightarrow & \sigma = \ln(E_f)/1.645. \end{array}$$

Uranie code to simulate a LogNormal random variable is:

```
TDataServer *tds = new TDataServer("tdssampler", "Sampler Uranie demo");
// using M, Ef and xmin
tds->addAttribute( new TLogNormalDistribution("ln", 1.2, 1.5, -0.5));
// to use ln(x) properties
// double mu = 0.5, sigma=1.; tds->setUnderlyingNormalParameters(mu,sigma);

TSampling *fsamp = new TSampling(tds, "lhs", 300);
fsamp->generateSample(); // Create a representative sample

tds->Draw("ln");
tds->Draw("log(ln)"); // Check that ln(ln) follows a normal law
```

Figure II.11 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

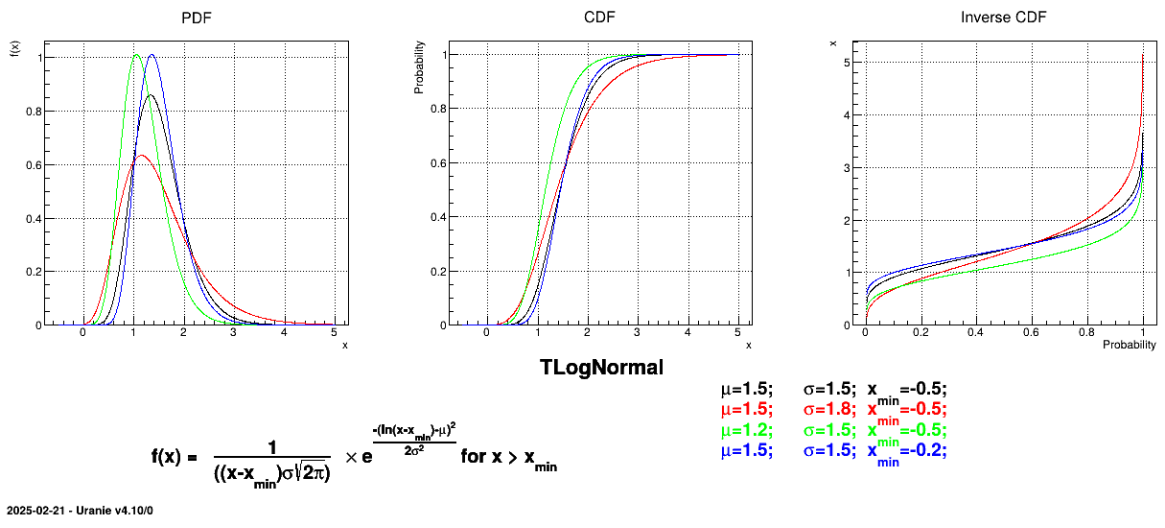


Figure II.11: Example of PDF, CDF and inverse CDF for LogNormal distributions.

Is it also possible to set boundaries to the infinite span of this distribution to create a truncated normal law. This can be done by calling the following method:

```
tds->getAttribute("ln")->setBounds(0.6,3.1); //truncate the law
```

The resulting PDF, CDF and inverse CDF, with and without truncation, can be seen, in this case, in Figure II.12 for a given set of parameters and various boundaries.

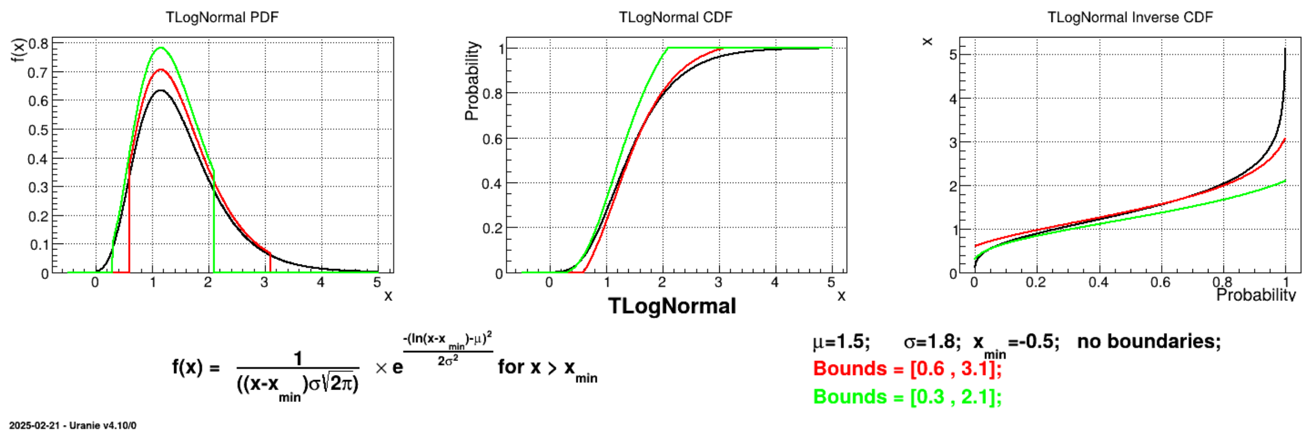


Figure II.12: Example of PDF, CDF and inverse CDF for a LogNormal truncated distribution.

II.2.5.7 Trapezium law

This law describes a trapezium whose large base is defined between a minimum and a maximum and its small base lies between a low and an up value, as

$$f(x) = \frac{2}{(x_{\text{up}} - x_{\text{low}}) + (x_{\text{max}} - x_{\text{min}})} \times Y$$

where $Y = 1$ for $x \in [x_{\text{low}}, x_{\text{up}}]$, $Y = \dfrac{(x - x_{\text{min}})}{(x_{\text{low}} - x_{\text{min}})}$ for $x \in [x_{\text{min}}, x_{\text{low}}]$ and $Y = \dfrac{(x_{\text{max}} - x)}{(x_{\text{max}} - x_{\text{up}})}$ for $x \in [x_{\text{up}}, x_{\text{max}}]$.

Uranie code to simulate a Trapezium random variable is:

```
TDataServer *tds = new TDataServer("tdssampler", "Sampler Uranie demo");
tds->addAttribute( new TTrapeziumDistribution("tr", 0.0, 1.0, 0.25, 0.75) );

TSampling *fsamp = new TSampling(tds, "lhs", 300);
fsamp->generateSample(); // Create a representative sample

tds->Draw("tr");
```

Figure II.13 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

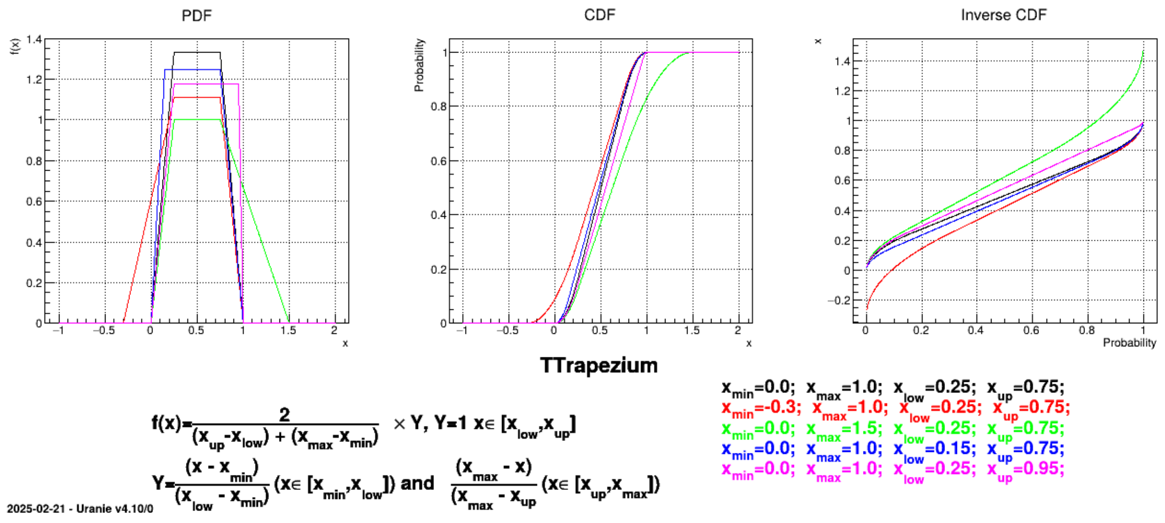


Figure II.13: Example of PDF, CDF and inverse CDF for Trapezium distributions.

II.2.5.8 UniformByParts law

The UniformByParts law is defined between a minimum and a median and between the median and a maximum, as

$$f(x) = \frac{0.5}{(x_{\text{med}} - x_{\text{min}})} \mathbb{I}_{[x_{\text{min}}, x_{\text{med}}]}(x) \qquad \text{and} \qquad f(x) = \frac{0.5}{(x_{\text{max}} - x_{\text{med}})} \mathbb{I}_{[x_{\text{med}}, x_{\text{max}}]}(x)$$

Uranie code to simulate a UniformByParts random variable is:

```
TDataServer *tds = new TDataServer("tdssampler", "Sampler Uranie demo");
tds->addAttribute( new TUniformByPartsDistribution("ubp", 0.0, 1.0, 0.5) );

TSampling *fsamp = new TSampling(tds, "lhs", 300);
fsamp->generateSample(); // Create a representative sample

tds->Draw("ubp");
```

Figure II.14 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

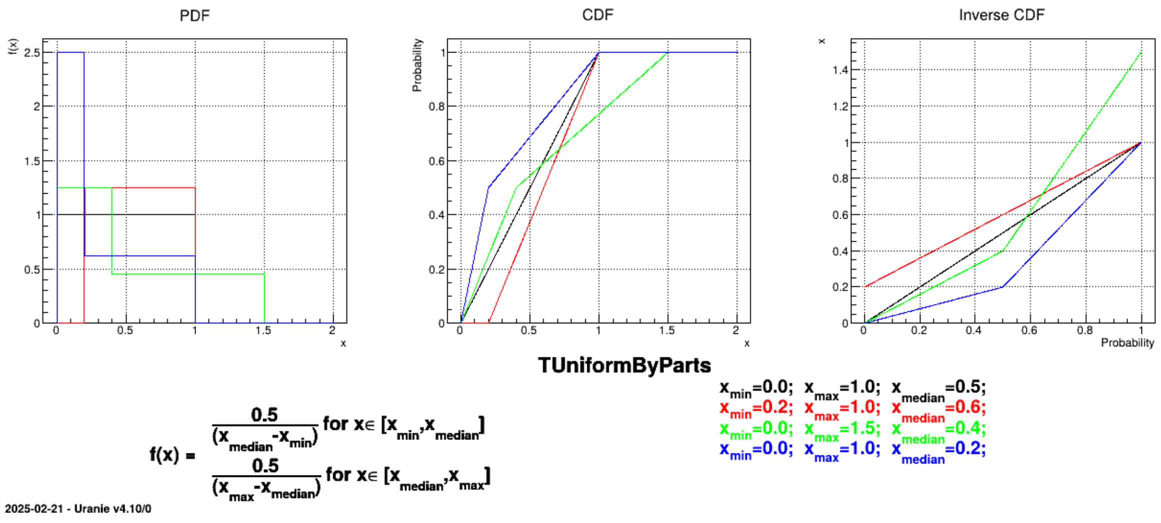


Figure II.14: Example of PDF, CDF and inverse CDF for UniformByParts distributions.

II.2.5.9 Exponential law

This law describes an exponential with a rate parameter λ and a minimum $x_{\min}$, as

$$f(x) = \lambda \times e^{-\lambda \times (x - x_{\min})} \, \mathbb{I}_{[x_{\min}, +\infty[}(x)$$

The rate parameter λ should be greater than 0.0001.

Uranie code to simulate an Exponential random variable is:

```
TDataServer *tds = new TDataServer("tdssampler", "Sampler Uranie demo");
tds->addAttribute( new TExponentialDistribution("exp", 0.5));

TSampling *fsamp = new TSampling(tds, "lhs", 300);
fsamp->generateSample(); // Create a representative sample

tds->Draw("exp");
```

Figure II.15 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

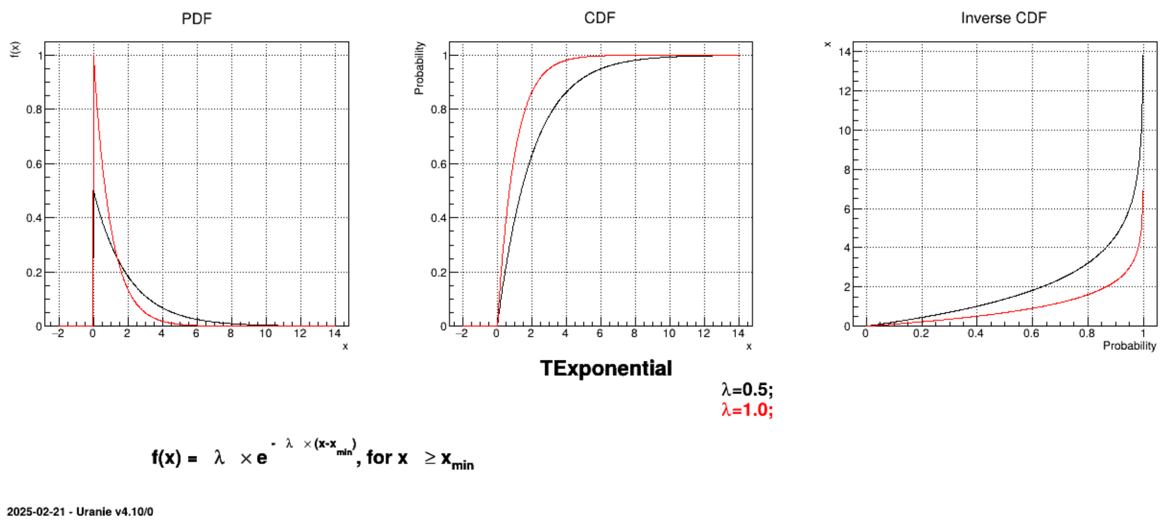


Figure II.15: Example of PDF, CDF and inverse CDF for Exponential distributions.

Is it also possible to set boundaries to the infinite span of this distribution to create a truncated Exponential law. This can be done by calling the following method:

```
tds->getAttribute("exp")->setBounds(0.4,6.0); //truncate the law
```

The resulting PDF, CDF and inverse CDF, with and without truncation, can be seen, in this case, in Figure II.16 for a given set of parameters and various boundaries.

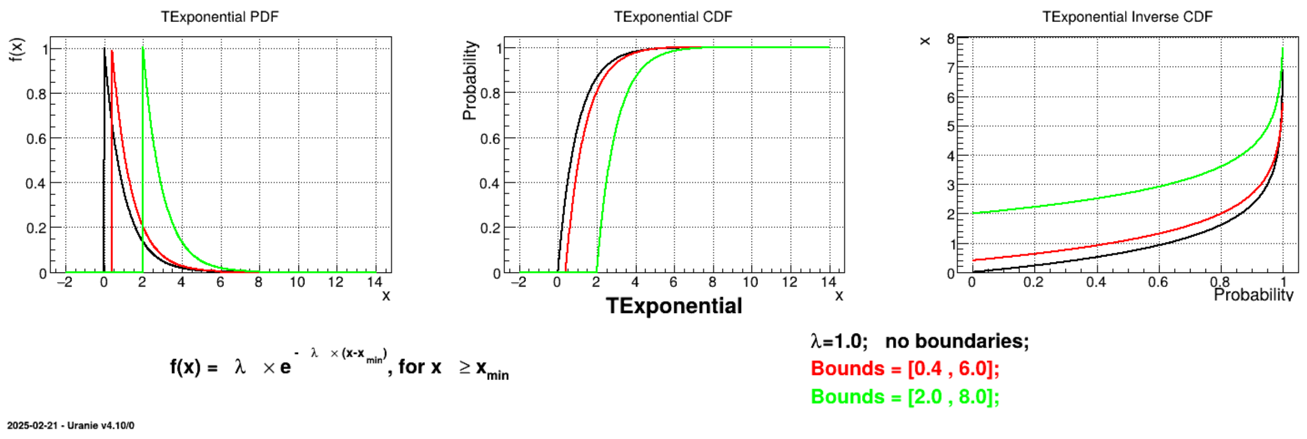


Figure II.16: Example of PDF, CDF and inverse CDF for a Exponential truncated distribution.

II.2.5.10 Cauchy law

This law describes a Cauchy-Lorentz distribution with a location parameter x_0 and a scale parameter γ , as

$$f(x) = \frac{\gamma}{\pi \times (\gamma^2 + (x - x_0)^2)}$$

The parameter γ should be greater than 0.0001.

Uranie code to simulate a Cauchy random variable is:

```
TDataServer *tds = new TDataServer("tdssampler", "Sampler Uranie demo");
tds->addAttribute( new TCauchyDistribution("cau", 0.3, 1.0));

TSampling *fsamp = new TSampling(tds, "lhs", 300);
fsamp->generateSample(); // Create a representative sample

tds->Draw("cau");
```

Figure II.17 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

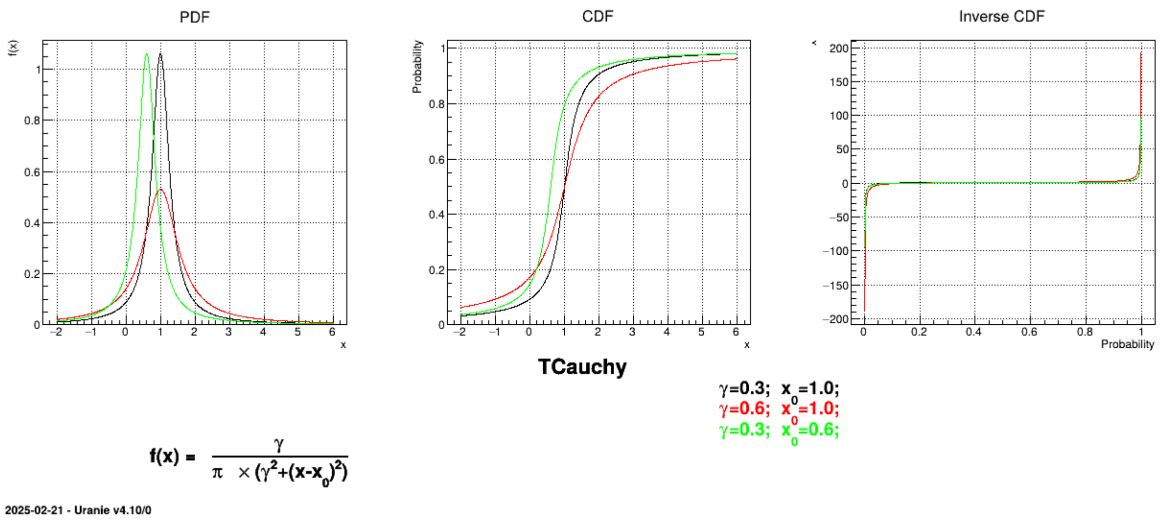


Figure II.17: Example of PDF, CDF and inverse CDF for Cauchy distributions.

Is it also possible to set boundaries to the infinite span of this distribution to create a truncated Cauchy law. This can be done by calling the following method:

```
tds->getAttribute("cau")->setBounds(-1.0,2.0); //truncate the law
```

The resulting PDF, CDF and inverse CDF, with and without truncation, can be seen, in this case, in Figure II.18 for a given set of parameters and various boundaries.

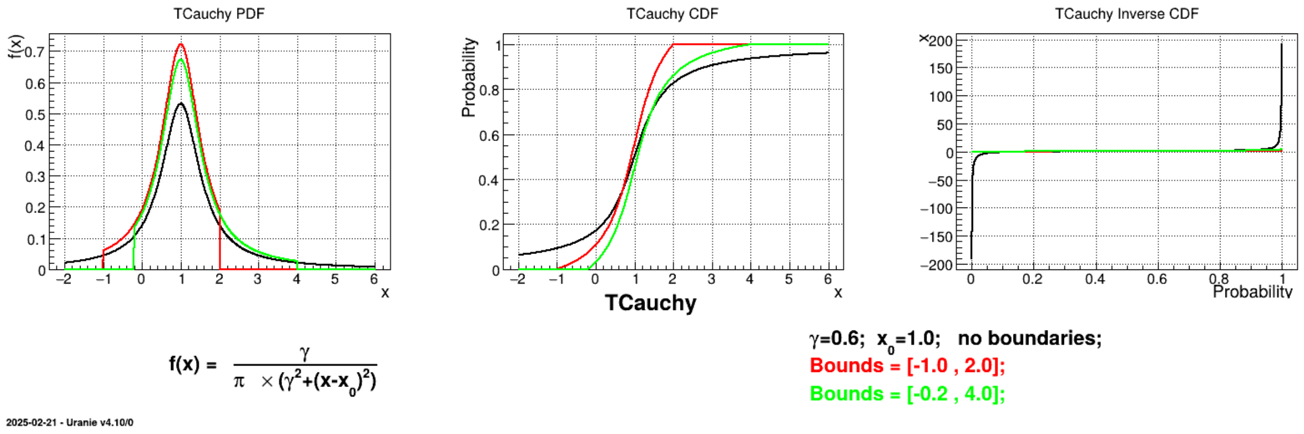


Figure II.18: Example of PDF, CDF and inverse CDF for a Cauchy truncated distribution.

II.2.5.11 GumbelMax law

This law describes a Gumbel max distribution depending on the mode μ and the scale β, as

$$f(x) = z \times \frac{e^{-z}}{\beta}, \text{ where } z = e^{\frac{-(x-\mu)}{\beta}}$$

The scale β should be greater than 0.000001 times μ

Uranie code to simulate a GumbelMax random variable is:

```
TDataServer *tds = new TDataServer("tdssampler", "Sampler Uranie demo");
tds->addAttribute( new TGumbelMaxDistribution("gm", 0.5, 2.0));

TSampling *fsamp = new TSampling(tds, "lhs", 300);
fsamp->generateSample(); // Create a representative sample

tds->Draw("gm");
```

Figure II.19 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

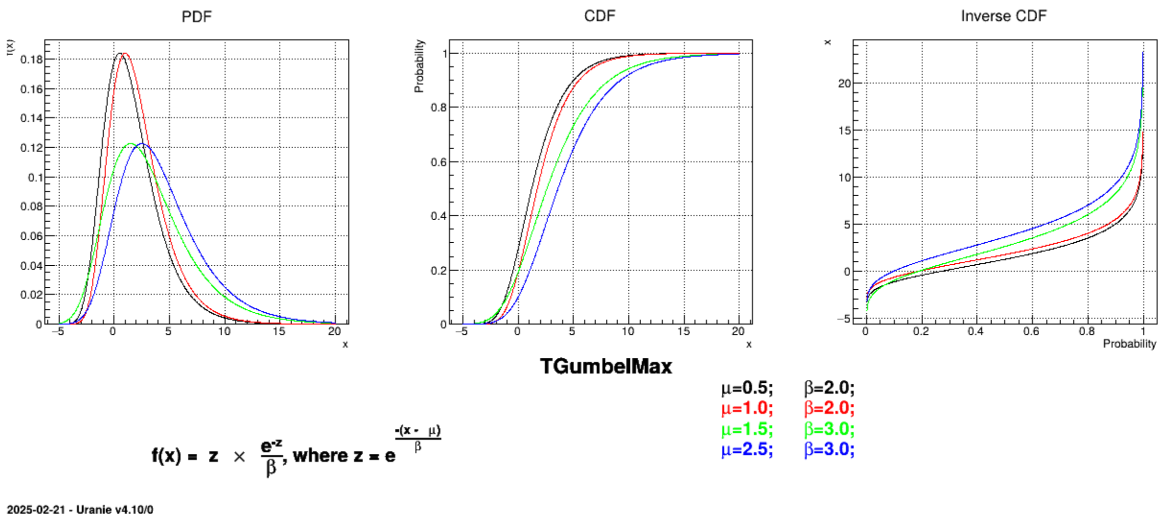


Figure II.19: Example of PDF, CDF and inverse CDF for GumbelMax distributions.

Is it also possible to set boundaries to the infinite span of this distribution to create a truncated GumbelMax law. This can be done by calling the following method:

```
tds->getAttribute("gm")->setBounds(-1.0,12.0); //truncate the law
```

The resulting PDF, CDF and inverse CDF, with and without truncation, can be seen, in this case, in Figure II.20 for a given set of parameters and various boundaries.

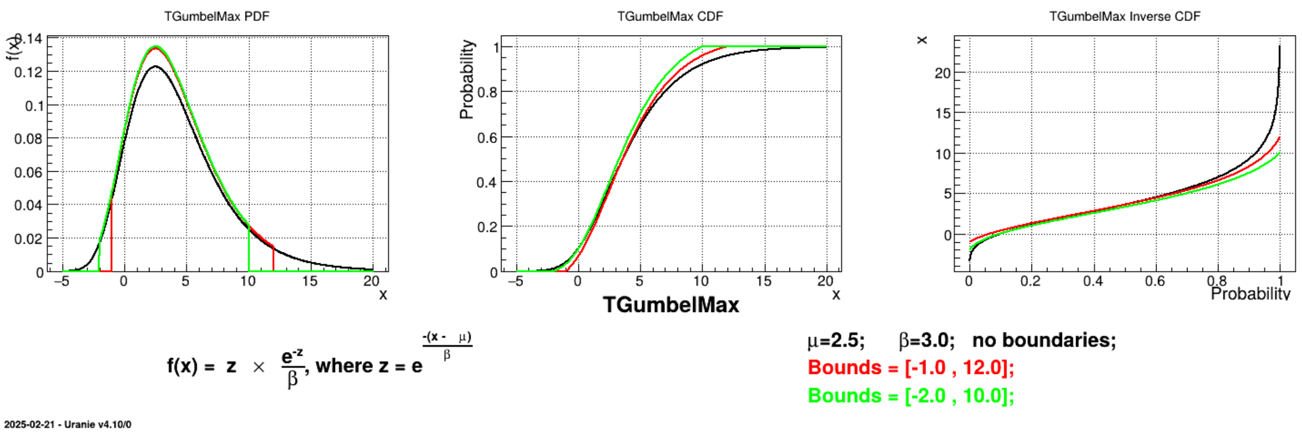


Figure II.20: Example of PDF, CDF and inverse CDF for a GumbelMax truncated distribution.

II.2.5.12 Weibull law

This law describes a weibull distribution depending on the location $x_{\min}$, the scale λ and the shape k, as

$$f(x) = \frac{k}{\lambda} \times \left(\frac{x - x_{\min}}{\lambda}\right)^{k-1} \times e^{-\left(\frac{x-x_{\min}}{\lambda}\right)^{k}} \mathbb{I}_{[x_{\min},+\infty[}(x)$$

Both λ and k should be greater than 0.0001.

Uranie code to simulate a Weibull random variable is:

```
TDataServer *tds = new TDataServer("tdssampler", "Sampler Uranie demo");
tds->addAttribute( new TWeibullDistribution("wei", 0.5, 2.0, -0.01) );

TSampling *fsamp = new TSampling(tds, "lhs", 300);
fsamp->generateSample(); // Create a representative sample

tds->Draw("wei");
```

Figure II.21 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

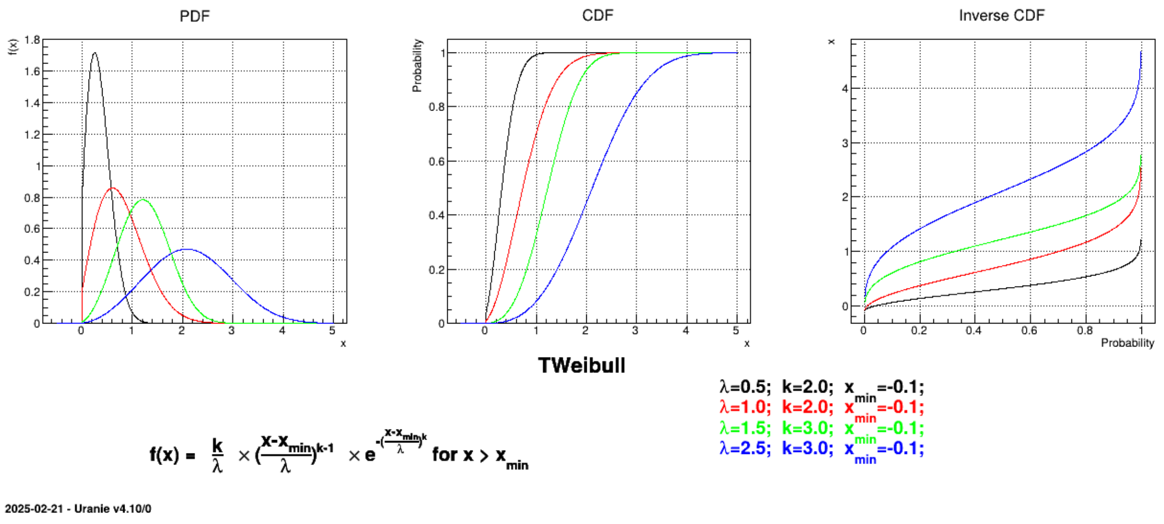


Figure II.21: Example of PDF, CDF and inverse CDF for Weibull distributions.

Is it also possible to set boundaries to the infinite span of this distribution to create a truncated Weibull law. This can be done by calling the following method:

```
tds->getAttribute("wei")->setBounds(0.2,1.8); //truncate the law
```

The resulting PDF, CDF and inverse CDF, with and without truncation, can be seen, in this case, in Figure II.22 for a given set of parameters and various boundaries.

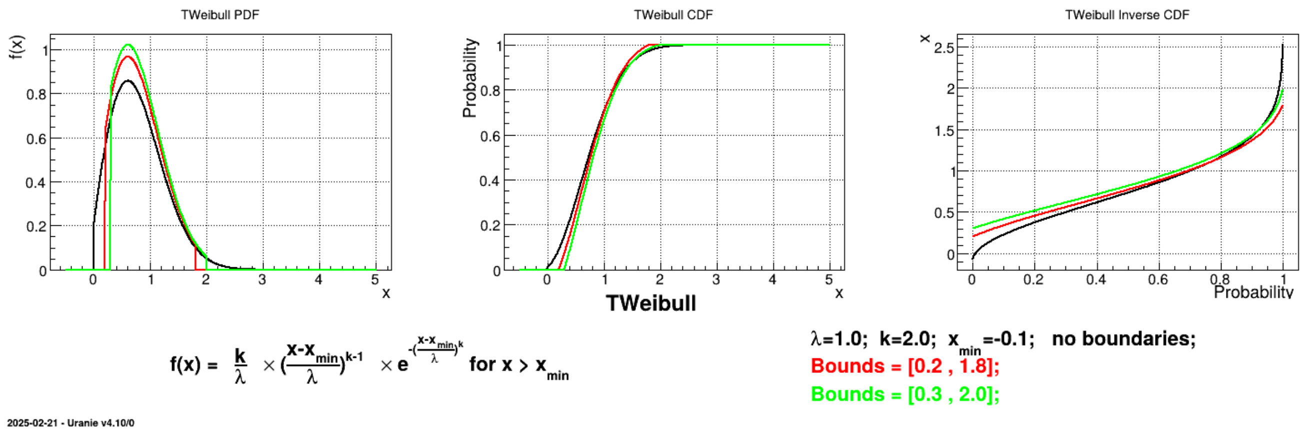


Figure II.22: Example of PDF, CDF and inverse CDF for a Weibull truncated distribution.

II.2.5.13 Beta law

Defined between a minimum and a maximum, it depends on two parameters α and β, as

$$f(x) = \frac{Y^{\alpha-1} \times (1-Y)^{\beta-1}}{B(\alpha,\beta)} \, \mathbb{1}_{[x_{\min},x_{\max}]}(x)$$

where $Y = \dfrac{(x - x_{\min})}{(x_{\max} - x_{\min})}$ and $B(\alpha,\beta)$ is the beta function. In the current implementation, both α and β must be greater than 0.0001.

Uranie code to simulate a Beta random variable is:

```
TDataServer *tds = new TDataServer("tdssampler", "Sampler Uranie demo");
tds->addAttribute( new TBetaDistribution("bet", 6.0, 6.0, 0.0, 2.0) );

TSampling *fsamp = new TSampling(tds, "lhs", 300);
fsamp->generateSample(); // Create a representative sample

tds->Draw("bet");
```

Figure II.23 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

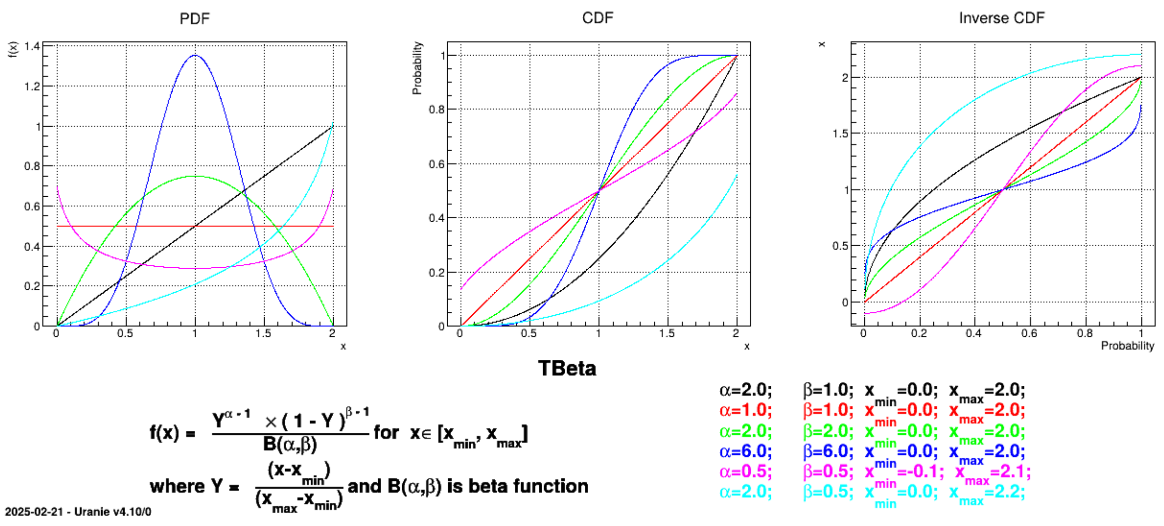


Figure II.23: Example of PDF, CDF and inverse CDF for Beta distributions.

II.2.5.14 GenPareto law

This law describes a generalised Pareto distribution depending on the location μ, the scale σ and a shape ξ, as

$$f(x) = \frac{1}{\sigma} \times \left(1 + \xi \left(\frac{x-\mu}{\sigma}\right)\right)^{-(1/\xi+1)}$$

In this formula, σ should be greater than 0.

Uranie code to simulate a GenPareto random variable is:

```
TDataServer *tds = new TDataServer("tdssampler", "Sampler Uranie demo");
tds->addAttribute( new TGenParetoDistribution("gpa", 1.0, 1.0, 0.3));

TSampling *fsamp = new TSampling(tds, "lhs", 300);
fsamp->generateSample(); // Create a representative sample

tds->Draw("gpa");
```

Figure II.24 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

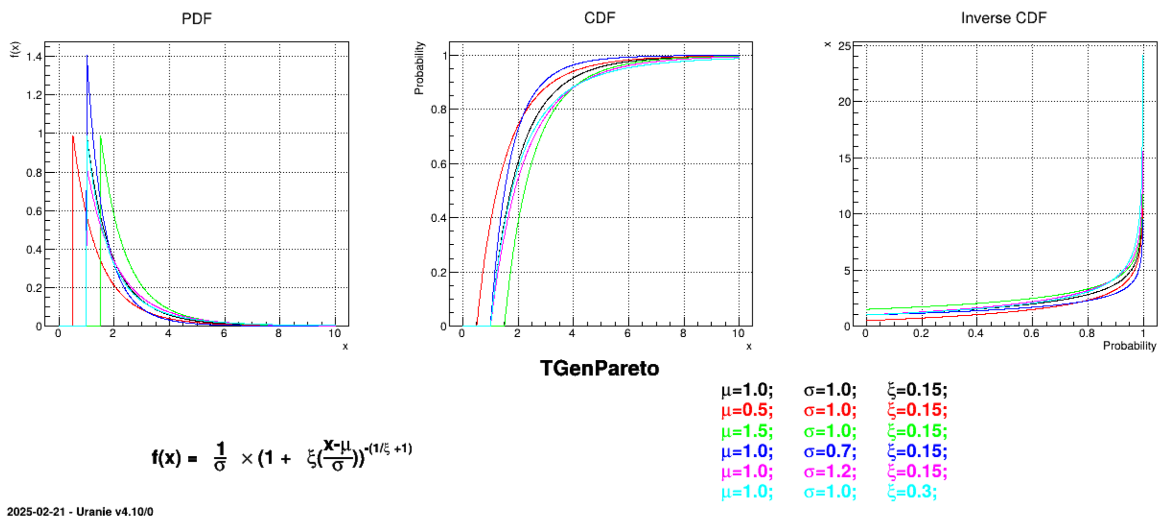


Figure II.24: Example of PDF, CDF and inverse CDF for GenPareto distributions.

Is it also possible to set boundaries to the infinite span of this distribution to create a truncated GenPareto law. This can be done by calling the following method:

```
tds->getAttribute("gpa")->setBounds(1.4,4.0); //truncate the law
```

The resulting PDF, CDF and inverse CDF, with and without truncation, can be seen, in this case, in Figure II.25 for a given set of parameters and various boundaries.

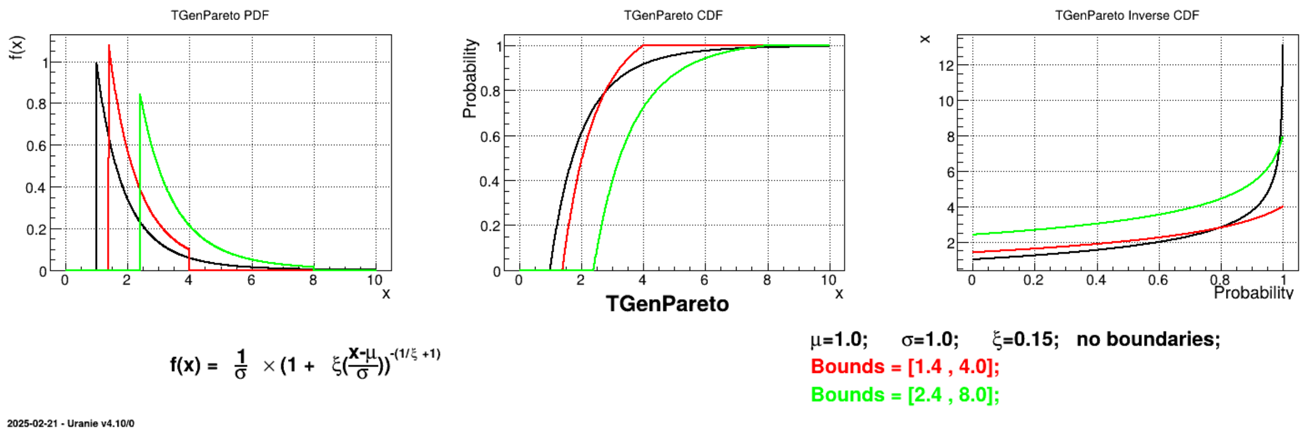


Figure II.25: Example of PDF, CDF and inverse CDF for a GenPareto truncated distribution.

II.2.5.15 Gamma law

The Gamma distribution is a two-parameter family of continuous probability distributions. It depends on a shape parameter α and a scale parameter β. The function is usually defined for x greater than 0, but the distribution can be shifted thanks to the third parameter called location (ξ) which should be positive. This parametrisation is more common in Bayesian statistics, where the gamma distribution is used as a conjugate prior distribution for various types of laws:

$$f(x) = \frac{(x-\xi)^{\alpha-1}e^{-(x-\xi)/\beta}}{\Gamma(\alpha)\beta^{\alpha}}\ \mathbb{I}_{]\xi,+\infty]}(x)$$

Uranie code to simulate a Gamma random variable is:

```
TDataServer *tds = new TDataServer("tdssampler", "Sampler Uranie demo");
tds->addAttribute( new TGammaDistribution("gam", 1.0, 2.0, 0.0)) ;

TSampling *fsamp = new TSampling(tds, "lhs", 300);
fsamp->generateSample(); // Create a representative sample

tds->Draw("gam");
```

Figure II.26 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

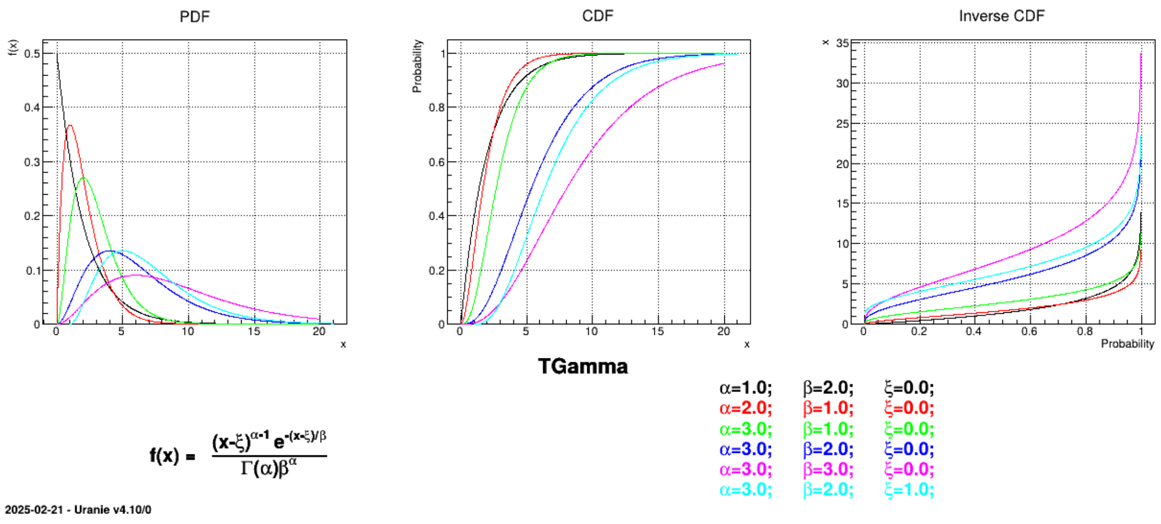


Figure II.26: Example of PDF, CDF and inverse CDF for Gamma distributions.

Is it also possible to set boundaries to the infinite span of this distribution to create a truncated Gamma law. This can be done by calling the following method:

```
tds->getAttribute("gam")->setBounds(0.1,1.6); //truncate the law
```

The resulting PDF, CDF and inverse CDF, with and without truncation, can be seen, in this case, in Figure II.27 for a given set of parameters and various boundaries.

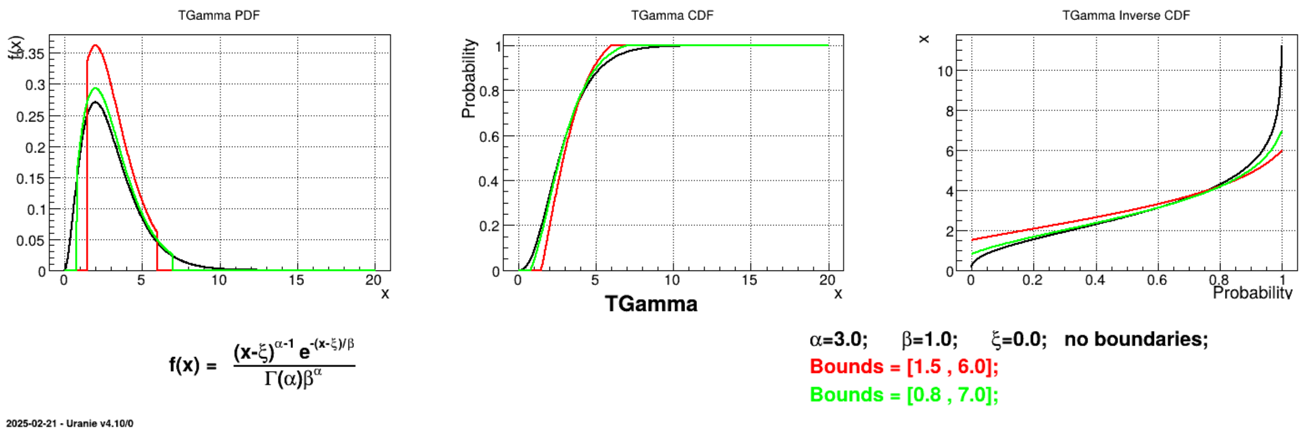


Figure II.27: Example of PDF, CDF and inverse CDF for a Gamma truncated distribution.

II.2.5.16 InvGamma law

The inverse-Gamma distribution is a two-parameter family of continuous probability distributions. It depends on a shape parameter α and a scale parameter β. The function is usually defined for x greater than 0, but the distribution can be shifted thanks to the third parameter called location (ξ) which should be positive.

$$f(x) = \frac{\beta^{\alpha}(x-\xi)^{-\alpha-1}e^{-\beta/(x-\xi)}}{\Gamma(\alpha)} \mathbb{I}_{]\xi,+\infty]}(x)$$

Uranie code to simulate a inverse-Gamma random variable is:

```
TDataServer *tds = new TDataServer("tdssampler", "Sampler Uranie demo");
tds->addAttribute( new TInvGammaDistribution("ing", 2.0, 0.5, 0.0));

TSampling *fsamp = new TSampling(tds, "lhs", 300);
fsamp->generateSample(); // Create a representative sample

tds->Draw("ing");
```

Figure II.28 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

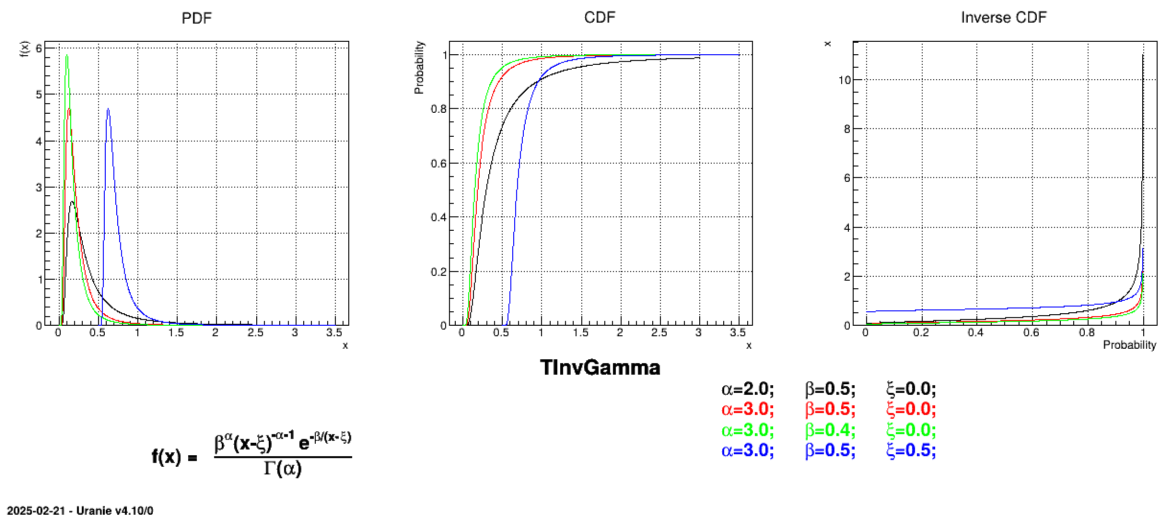


Figure II.28: Example of PDF, CDF and inverse CDF for InvGamma distributions.

Is it also possible to set boundaries to the infinite span of this distribution to create a truncated InvGamma law. This can be done by calling the following method:

```
tds->getAttribute("ing")->setBounds(-3.0,8.0); //truncate the law
```

The resulting PDF, CDF and inverse CDF, with and without truncation, can be seen, in this case, in Figure II.29 for a given set of parameters and various boundaries.

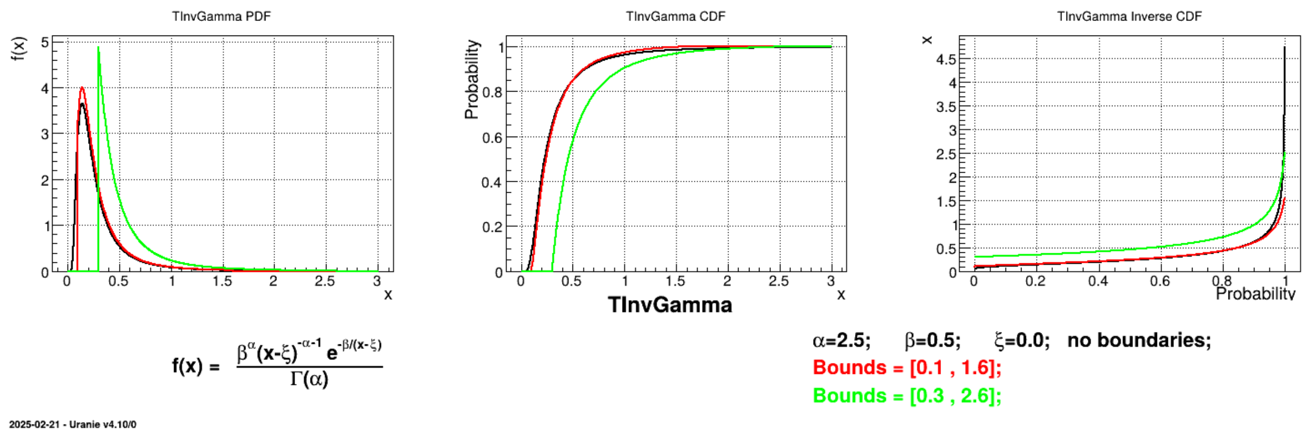


Figure II.29: Example of PDF, CDF and inverse CDF for a InvGamma truncated distribution.

II.2.5.17 Student Law

Warning
This distribution is available only if the ROOT "mathmore" feature has been installed when your ROOT version was brought (you can check this by running `root-config --has-mathmore`. If not found, this law cannot be used.

The Student law is simply defined with a single parameter: the degree-of-freedom (**DoF**). The probability density function is then set as

$$f(x) = \frac{1}{\sqrt{k\pi}} \frac{\Gamma\left(\frac{k+1}{2}\right)}{\Gamma\left(\frac{k}{2}\right)} \left(1+\frac{t^2}{k}\right)^{-\frac{k+1}{2}}$$

where Γ is the Euler's gamma function.

Uranie code to simulate an student random variable is:

```
TDataServer *tds = new TDataServer("tdssampler", "Sampler Uranie demo");
tds->addAttribute( new TStudentDistribution("stu", 5));

TSampling *fsamp = new TSampling(tds, "lhs", 300);
fsamp->generateSample(); // Create a representative sample

tds->Draw("stu");
```

Figure II.30 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

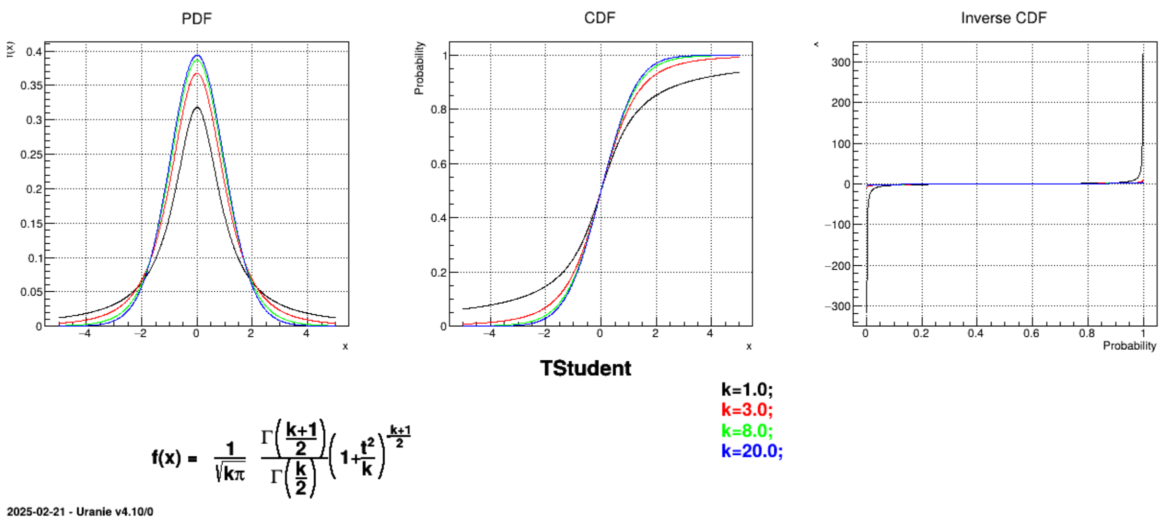


Figure II.30: Example of PDF, CDF and inverse CDF for Student distribution.

Is it also possible to set boundaries to the infinite span of this distribution to create a truncated Student law. This can be done by calling the following method:

```
tds->getAttribute("stu")->setBounds(-1.4,2.0); //truncate the law
```

The resulting PDF, CDF and inverse CDF, with and without truncation, can be seen, in this case, in Figure II.31 for a given set of parameters and various boundaries.

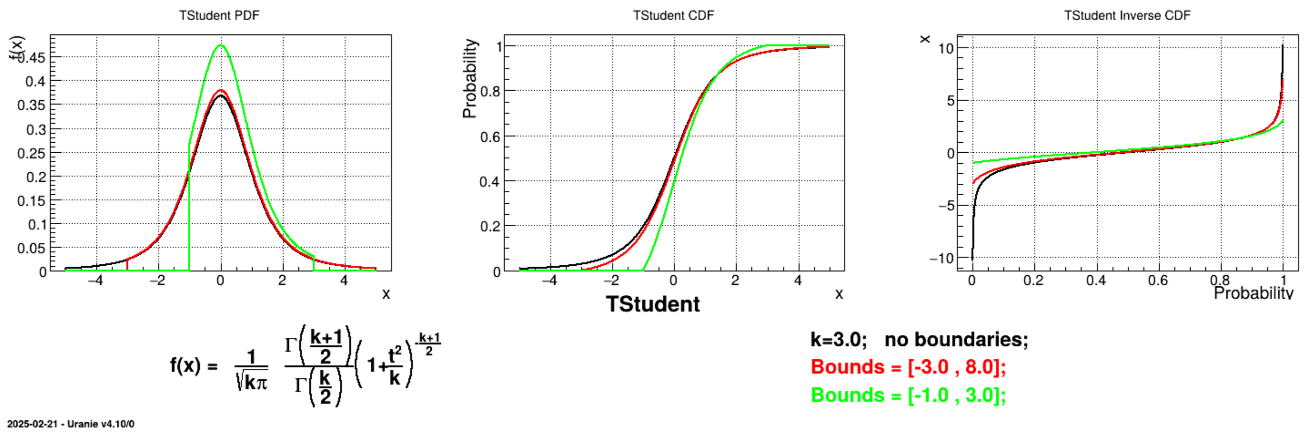


Figure II.31: Example of PDF, CDF and inverse CDF for a Student truncated distribution.

II.2.5.18 Generalized normal law

This law describes a generalized normal distribution depending on the location μ, the scale α and the shape β, as

$$f(x) = \frac{\beta}{2\alpha\Gamma(1/\beta)} \times \times e^{-\left(\frac{x-\mu}{\alpha}\right)^{\beta}}$$

Both α and β should be greater than 0.

Uranie code to simulate a generalized normal random variable is:

```
TDataServer *tds = new TDataServer("tdssampler", "Sampler Uranie demo");
tds->addAttribute( new TGeneralizedNormalDistribution("gennor", 0.0, 1.0, 3.0) );

TSampling *fsamp = new TSampling(tds, "lhs", 300);
fsamp->generateSample(); // Create a representative sample

tds->Draw("gennor");
```

Figure II.32 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

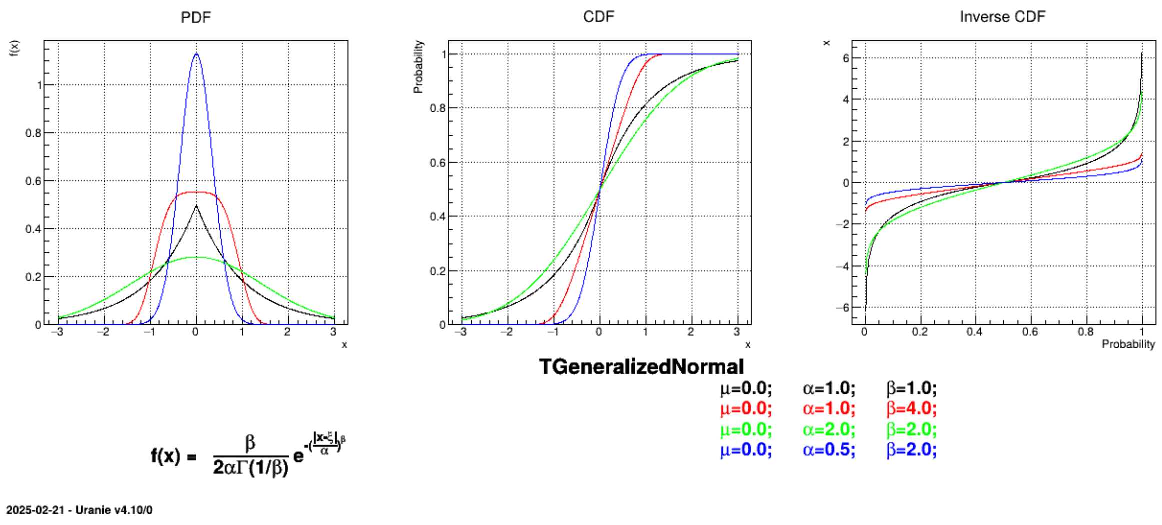


Figure II.32: Example of PDF, CDF and inverse CDF for generalized normal distributions.

Is it also possible to set boundaries to the infinite span of this distribution to create a truncated generalized normal law. This can be done by calling the following method:

```
tds->getAttribute("gennor")->setBounds(-0.8,1.6); //truncate the law
```

The resulting PDF, CDF and inverse CDF, with and without truncation, can be seen, in this case, in Figure II.33 for a given set of parameters and various boundaries.

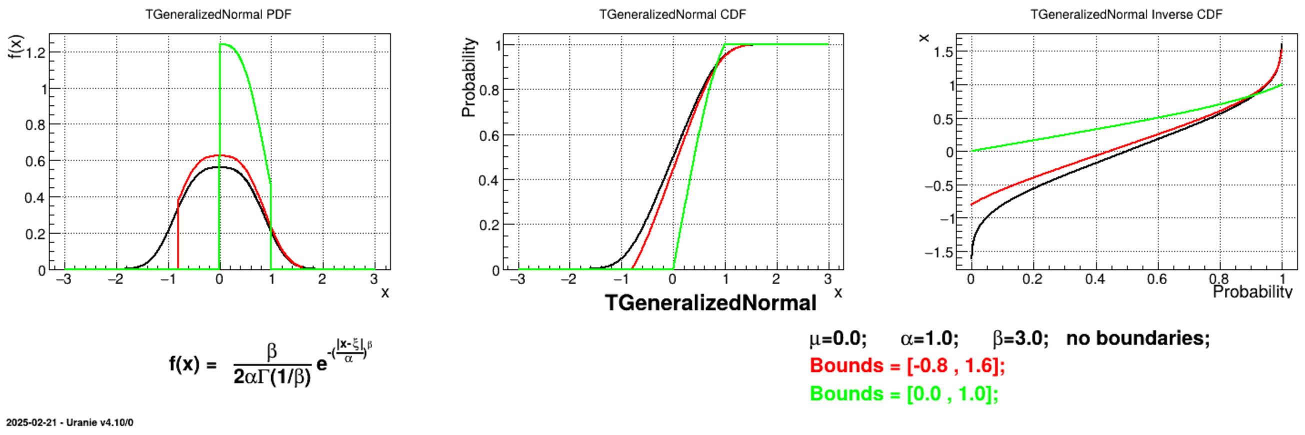


Figure II.33: Example of PDF, CDF and inverse CDF for a generalized normal truncated distribution.

II.2.5.19 Composing law

It is possible to imagine a new law, hereafter called *composed law*, by combining different pre-existing laws in order to model a wanted behaviour. This law would be defined with N pre-existing laws whose densities are noted $\{f_j\}_{1\leq j\leq N}$, along with their relative weights $\{\omega_j\}_{1\leq j\leq N} \in (\mathbb{R}^+)^N$ and the resulting density is then written as

$$f(x) = \sum_{j=1}^{N} \omega_j f_j(x).$$

Uranie code to simulate a composition of three normally-distributed laws (with their own statistical properties):

```
TDataServer *tds = new TDataServer("tdssampler", "Sampler Uranie demo");
TComposedDistribution *comp = new TComposedDistribution("compo");
comp->addDistribution(new TNormalDistribution("n1", -1.5, 0.2), 1.2);
comp->addDistribution(new TNormalDistribution("n2", 0, 0.8), 1.0);
comp->addDistribution(new TNormalDistribution("n3", 1.5, 0.2), 0.8);
tds->addAttribute(comp);

TSampling *fsamp = new TSampling(tds, "lhs", 300);
fsamp->generateSample(); // Create a representative sample

tds->Draw("compo");
```

Figure II.34 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

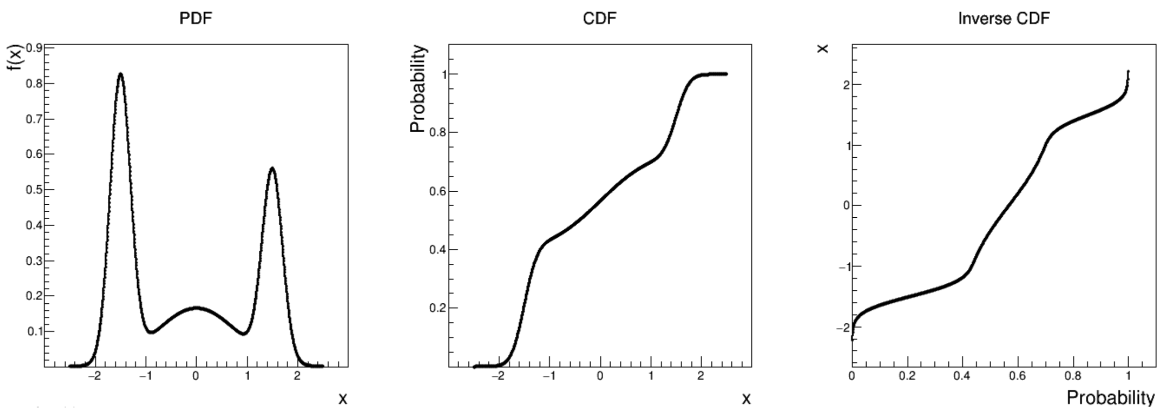


Figure II.34: Example of PDF, CDF and inverse CDF for a composed distribution made out of three normal distributions with respective weights.

Is it also possible to set boundaries to the infinite span of this distribution, if it is created from at least one infinite-based law, to create a truncated composed law. This can be done by calling the following method:

```
tds->getAttribute("compo")->setBounds(-1.6,1.8); //truncate the law
```

The resulting PDF, CDF and inverse CDF, with and without truncation, can be seen, in this case, in Figure II.35 for a given set of parameters and various boundaries.

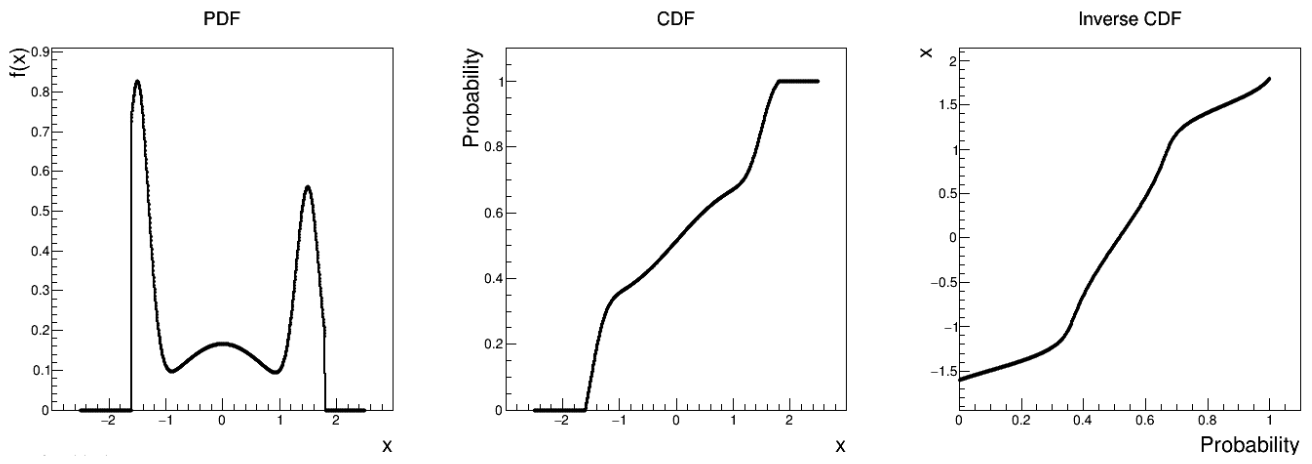


Figure II.35: Example of PDF, CDF and inverse CDF for a truncated composed distribution made out of three normal distributions with respective weights.

The only specific method that is new for the composition is the `addDistribution` method whose signature is the following one:

```
int addDistribution(URANIE::DataServer::TStochasticAttribute *statt, double weight=1.);
```

The first element is a pointer to a `TStochasticAttribute` (so any object that is an instance of a class that derives form it). The second one is the weight (which is 1 by default) and which is the ω constant written in the formula above.

Warning The theoretical element (mean, standard deviation and mode) can not always be measured for certain stochastic distribution (see the Cauchy's one for instance).

- If one wants to add such a distribution in a composed law, a warning exception should pop-up to state that theoretical properties can not be estimated.
- As for the mode, several distributions prevent from having a single-point mode estimation (for instance the Uniform distribution). The mode estimation should then be taken with great care.

II.3 Data handling

This section describes the data import from an **ASCII** flat file in a `TDataServer` of Uranie using the formalism of **Salome** tables and JSON format. In both cases, the **ASCII** file is composed of 2 parts: the **header** and the **experiment matrix**.

- The **header** describes the information related to the database and related to its attributes:
  - the name of the database *(optional)*;
  - the title of the database *(optional)*;
  - the date of the database's saving *(optional)*;
  - the name of variables *(mandatory if the title/label's not specified)*. This information will enable to have access to theses variables either to transform them or to produce graphs. The chosen name has to be explicit but rather **short** (smaller than 50 characters);

- the title/label of variables *(mandatory if the name of variables is not specified)*. This information will be the axis tick marks when this variable is visualised on the plot. As it is possible to use LaTeX commands, the *name* (must be explicit and short) is distinguished from the *label* so as to obtain a good graph rendering.
- the variable units *(optional)*. This information aims at improving the graph rendering (the unit being displayed next to the label).
- the nature of the variables *(optional unless dealing with new types)*. This information is not mandatory if one wants to handle only double-precision variables. The possible value being "D" for double, "S" for string and "V" for vectors.

• The second part is the *numerical* experiment matrix.

Using this formalism, it is enough to provide the name of attributes of the ASCII file in order to load the file in Uranie.

II.3.1 Main format of input/output

II.3.1.1 The Salome-table format

This is the main format used throughout the history of the Uranie-platform. The different header information are set thanks to keywords. A header line begins with the character **'#'** followed by a keyword characterising the type of information and by the **':'** character . Then, the information are separated by the '|' character. The list of keywords is:

Keywords	Description
NAME	The name of the database
TITLE	The title of the database
DATE	The date of saving (only towards writing or export)
COLUMN_NAMES	The names of attributes
COLUMN_TYPES	The natures of attributes
COLUMN_TITLES	The titles of attributes
COLUMN_UNITS	The units of attributes

Table II.2: List of keywords of *header* in ASCII files.

An example is the file `"geyser.dat"`in the *data* directory of installation of **Uranie** (`$URANIESYS/share/uranie/macros/`

```
> more $URANIESYS/share/uranie/macros/geyser.dat

#NAME: geyser
#TITLE:  geyser data
#COLUMN_NAMES: x1| x2
#COLUMN_TITLES: x_{1}| "#delta P_{#sigma}"
#COLUMN_UNITS: Sec^{-1}| bar

3.600 79.000
1.800 54.000
3.333 74.000
2.283 62.000
4.533 85.000
2.883 55.000
4.700 88.000
3.600 85.000
1.950 51.000
4.350 85.000
1.833 54.000
```

```
3.917 84.000
4.200 78.000
1.750 47.000
4.700 83.000
2.167 52.000
...
```

Uranie accepts several forms of file endings: it is possible the file ends with a white line or with a line with empty spaces, but also to end just after the last data.

Uranie does not accept data with "holes" (empty lines) like as follows in this modified version of the *"geyser.dat"* file:

```
#NAME: geyser
#TITLE:  geyser data
#COLUMN_NAMES: x1| x2
#COLUMN_TITLES: x_{1}| "#delta P_{#sigma}"
#COLUMN_UNITS: Sec^{-1}| bar

3.600 79.000
1.800 54.000
3.333 74.000
2.283 62.000
4.533 85.000

2.883 55.000
4.700 88.000
3.600 85.000
1.950 51.000
4.350 85.000
1.833 54.000
3.917 84.000
4.200 78.000
1.750 47.000
4.700 83.000
2.167 52.000
...
```

In this case, **Uranie** considers that data processing ends with the white line located in the middle of the data lines. This would be equivalent to use the following data:

```
#NAME: geyser
#TITLE:  geyser data
#COLUMN_NAMES: x1| x2
#COLUMN_TITLES: x_{1}| "#delta P_{#sigma}"
#COLUMN_UNITS: Sec^{-1}| bar

3.600 79.000
1.800 54.000
3.333 74.000
2.283 62.000
4.533 85.000
```

Tip

Only the line associated to the keyword **COLUMN_NAMES** is mandatory except if **COLUMN_TITLES** is specified. Moreover, the keyword itself is also optional; the next line is correct *# x1| x2* to specify both variables of the geyser data.

Warning
An empty line MUST be kept between the header and data matrix .

Particular case of strings and vector

The following example shows how to precise the content of vectors and strings if such information have to be read. In this case, the field `#COLUMN_TYPES:` is mandatory and the way it works is equivalent to the column name one (the delimiter is the "|" sign) but it needs only one letter to define the type. Apart from that, the string can be written as it comes **as long as it does not contains blanks (!!)**, while the vectors values are dump with the same format as the double-precision one, using a comma as separator. The following file is a correct input file for a `TDataServer`

```
#COLUMN_NAMES: day|place|hour|guest_list|food
#COLUMN_TYPES: D|S|D|V|S

5 restaurant 4 2,3,4,5 chocolate
21 home 3 6,1,8,4,3 almond
```

The example shown above is working properly as there is no problematic behaviour in the data. Handling strings and vectors can however be tricky as they respectively can be an empty string and an empty vector. This would result in a missing number of field in a specific line which will make the `fileDataRead` crashed. To avoid this, all the files used in the Launcher and Relauncher (and the Salome-table discussed here as well) contains properties specific to both vectors and strings:

- String properties: a character can be specified as begin and end for dumping and reading purpose. The ones chosen by default for the Salome-table format shown here being the double-quote sign ".
- Vector properties: a character can be specified as begin, end and delimiter for dumping and reading purpose. The ones chosen by default for the Salome-table format shown here being respectively [,] and the commas.

This results in the fact, that the following file gives the exact same dataserver as the one shown previously. It is actually the style chosen when calling the `exportData` method of a `TDataServer` to allow the user to handle empty strings and vectors if wanted.

```
#COLUMN_NAMES: day|place|hour|guest_list|food
#COLUMN_TYPES: D|S|D|V|S

5 "restaurant" 4 [2,3,4,5] "chocolate"
21 "home" 3 [6,1,8,4,3] "almond"
```

II.3.1.2 The JSON format

Brought in version 3.9, the format has been implemented as it is broadly used to transmit data in a very simple way. A choice has been made on the way the header are displayed: a "_metadata" field is compulsory inside which the list of flag is gathered in Table II.3.

The second part that provides the data itself, looks alike a key-value table and handles easily all the attribute types. Here is an example of file with the "geyser" data content, shown previously:

```
{
  "_metadata" :
  {
    "_comment" : "CurrentComment",
```

Keywords	Description
table_name	The name of the database
table_description	The title of the database
date	The date of saving (only towards writing or export)
short_names	The names of attributes
types	The natures of attributes
long_names	The titles of attributes
units	The units of attributes

Table II.3: List of keywords of *header* in ASCII files.

```
    "date" : "Fri Oct 28 10:41:44 2016",
    "short_names" : [ "x1", "x2", "geyser__n__iter__" ],
    "table_description" : "Les donnees du geyser",
    "table_name" : "geyser",
    "types" : [ "D", "D", "D" ],
    "units" : [ "Sec", "", "" ]
  },
  "items" :
  [

    {
      "geyser__n__iter__" : 1.0,
      "x1" : 3.60,
      "x2" : 79.0
    },

    {
      "geyser__n__iter__" : 2.0,
      "x1" : 1.80,
      "x2" : 54.0
    },



    {
      "geyser__n__iter__" : 272.0,
      "x1" : 4.4670,
      "x2" : 74.0
    }
  ]
}
```

II.3.2 Import data from an ASCII file

An example of import of the data file "`geyser.dat`" (available in the Uranie-macros folder) is shown below leading to a 2D scatterplot of the variable **x2** versus the variable **x1**

```
using namespace URANIE::DataServer; ❶[1]

TDataServer * tdsGeyser = new TDataServer("tdsgeyser", "Geyser database"); ❷[2]
tdsGeyser->fileDataRead("geyser.dat");❸[3]
tdsGeyser->draw("x2:x1");❹[4]
```

Description of the import of an ASCII file

❶ Setting the namespace. This instruction is useless when the provided rootlogon has been loaded as all Uranie-namespaces have been loaded as well.

❷ Defining a pointer *tdsGeyser* to an object of type `TDataServer` whose name is *"tdsgeyser"* and whose title is *"Geyser database"*. These information are used by the `export` or `printLog` methods.

❸ Loading data contained in an ASCII file `$URANIESYS/share/uranie/macros/geyser.dat`.

❹ Plot of the scatterplot of the variable **x2** versus the variable **x1**

The obtained graph is the following:

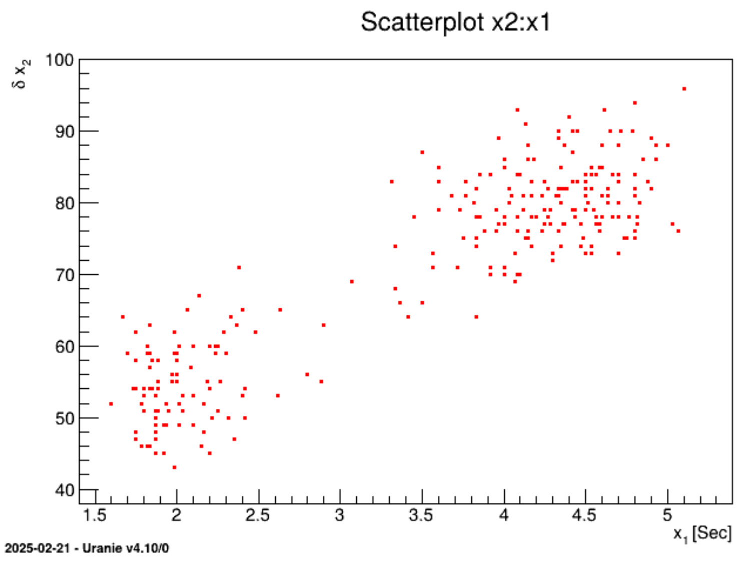


Figure II.36: Import data from an ASCII file

Various examples of macros loading data in a `TDataServer` with different treatment applied on, are provided in the use-case chapter of this user manual, between Section XIV.2.3 and Section XIV.2.7.

Summary: Loading data (ASCII file)

- `fileDataRead` (**TString** filename, **bool** saveTuple=true, **bool** preAddReload=false)

  Loads the data contained in the ASCII file ("Salome-table" format). The possible arguments are:

  – The name of the file (compulsory)

  – A boolean to state whether the file should be saved as root-ntuple once the reading is complete (not mandatory, set to true by default)

  – A boolean to allow pre-adding or reloading a database (advanced, not recommended for beginners). This allows

    * Pre-adding variables: very technical. It should not be used unless discussed with developers.
    * Reloading database: Allow to reload a database file without complaining, AS long AS the provided file exactly matches the list of attribute in the `TDataServer` object

- `fileDataReadJSon` (**TString** filename, **bool** saveTuple=true)

  Loads the data contained in the ASCII file (JSON format). The possible arguments are:

  – The name of the file (compulsory)

  – A boolean to state whether the file should be saved as root-ntuple once the reading is complete (not mandatory, set to true by default)

II.3.3 Import data from a TNtuple/TDSNtuple/TTree

From a `TTree` object (or any of its derived-object) contained in a ROOT-file, it is possible to import data, with or without selection of a variable and addition of other ones through formula then deletions of patterns ensuring a criterion

```
using namespace URANIE::DataServer; ❶[1]

TDataServer * tds = new TDataServer(); ❷[2]
tds->ntupleDataRead("hsimple.root","ntuple","px*px:*:py*py","px*px+py*py<2.0"); ❸[3]
tds->draw("py:px"); ❹[4]
```

Description of data importation of a TTree from a ROOT file

❶ Specification of the namespace.

❷ Creation of a pointer *tds* to an object of type `TDataServer` without name and title.

❸ Fill the `TDataServer` with all branches contained in the tree *ntuple* that is contained in `hsimple.root`. Two new attributes are computed on the fly and a selection is applied on the patterns to be kept;

❹ Plots scatterplot of the variable **py** versus the variable **px**

In this case, *tds* is constructed from the `TTree` *ntuple* contained in the file `hsimple.root` where all initial variables are kept (* character). Figure II.37 shows the content of the *ntuple*. Two new variables are then added on top, defined by the equations "px*py" and "py*px" on both side of the "*" string. A cut is finally done, to exclude all data that would satisfy the following equation $px^2 + py^2 < 2$

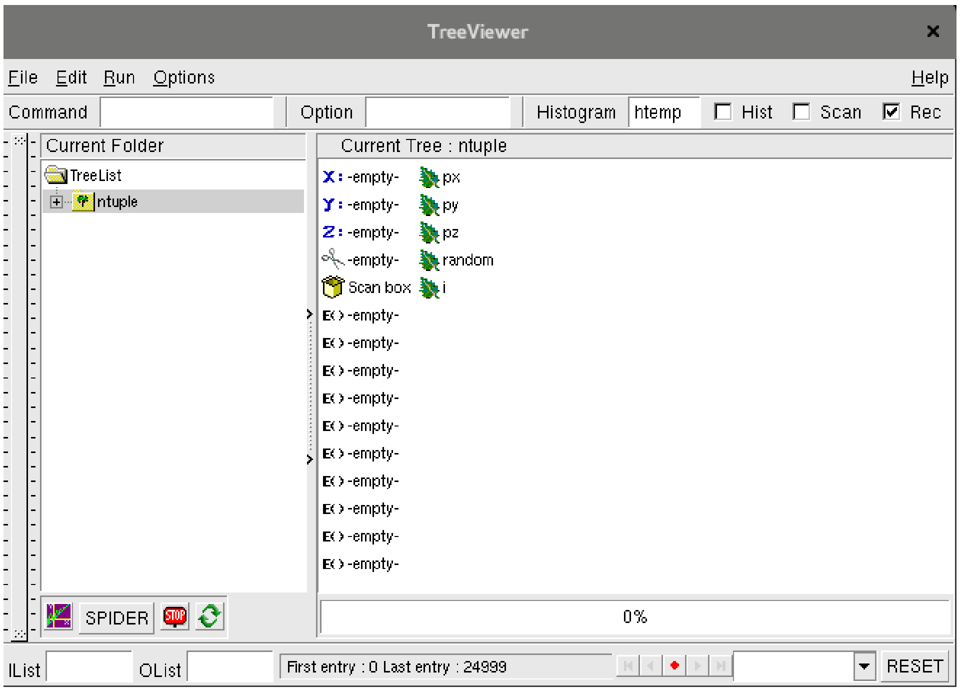


Figure II.37: Content of the *ntuple* tree contained in "hsimple.root" file.

The obtained graph is as follow:

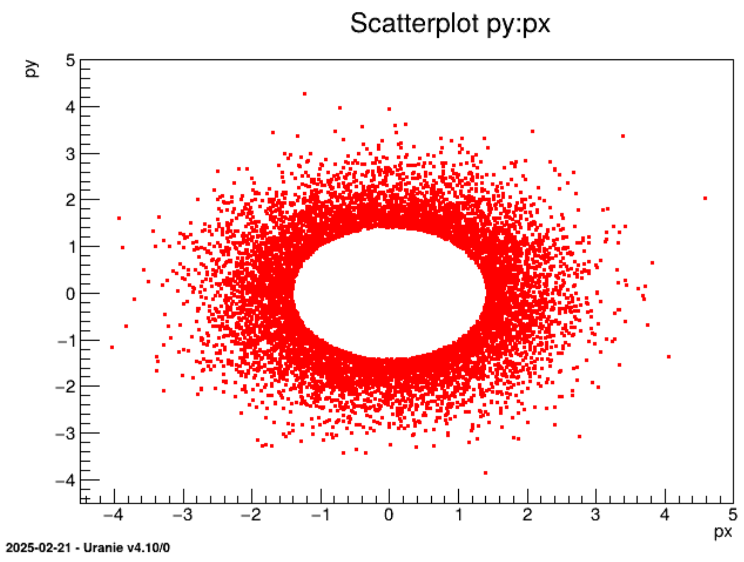


Figure II.38: Data importation from a TNtuple

Summary: Loading data (TNtupleD of ROOT in a TFile)

- `ntupleDataRead` (**const char** * file, **const char** * tree, **const char** * svar="*", **const char** * cut="",)

  Loads the data contained in the `TNtupleD tree` of file ROOT `TFile file` by selecting/creating the list of variables `svar` (the variables are separated by the ":" character, and "*" character means recalling all variables of the TNtupleD) and by deleting all patterns ensuring the criterion `cut`.

II.3.4 Adding attributes to a `TDataServer`

Attributes can always be added to an existing `TDataServer` object, whether it is *empty* (just after its constructor) or *not* (after the data loading from either an ASCII file, or a TTree or a database of type **SQL**). A simple example is provided and decomposed in Section XIV.2.1.

First of all, let us consider the case of an empty `TDataServer`. We add attributes using the method `addAttribute(` **TAttribute** *att).

```
TDataServer * tds = new TDataServer("tds", "new TDataServer");
tds->addAttribute( new TAttribute("x1"));            ❶[1]
tds->addAttribute( new TAttribute("x2", 2.5, 5.));  ❷[2]
tds->addAttribute( "x3" );                           ❸[3]
tds->addAttribute( "x4", 2.5, 5.);                   ❹[4]
```

Description of attributes adding to an empty TDataServer

❶ Adding a new attribute **x1** to the TDataServer from a `TAttribute` with a name (minimal constructor).

❷ Adding a new attribute **x2** to the TDataServer from a `TAttribute` with a name and the extreme values (minimal and maximal).

❸ Equivalent to previous one: adding a new attribute **x3** to the TDataServer just by giving its name (minimal constructor).

❹ Equivalent to previous one: adding a new attribute **x4** to the TDataServer by giving its name and the extreme values (minimal and maximal).

The difference between the methods with `new` in it and the others, is basically arising from the way one handles the memory. The last ones (3 and 4) allow the user not to worry about anything, while, in the case of implementation 1 and 2, one should be aware that every `new` should imply at some point a `delete`. For most user, this is not of utmost importance as usual scripts would contain very few `new` command and no loop. If this is not the case (for instance if one does have loop and many object creation in it) do not hesitate to contact the Uranie-team to prevent any slowing down of the code.

The specification of a `TAttribute` is further detailed in Section II.2.

We can define new attributes using mathematical expressions with respect to other existing attributes. The name and the mathematical expression are the only mandatory arguments; its title and unit can also be precised, but both arguments are optional.

```
TDataServer * tdsGeyser = new TDataServer("tdsgeyser", "Geyser DataSet");
tdsGeyser->fileDataRead("geyser.dat");
tdsGeyser->addAttribute("cd1","sqrt(x2) * x1");    ❶[1]
tdsGeyser->addAttribute("cd2","sqrt(x2*x1)","#Delta p_{#sigma}","sec^{-1}");  ❷[2]

tdsGeyser->draw("cd2:cd1");    ❸[3]
```

Description attribute adding to a TDataServer from formulas

❶ Adding a new attribute **cd1** to the TDataServer defined by a mathematical expression as a function of *x1* and *x2* attributes:

$$cd1 \;=\; x_1 * \sqrt{x_2}$$

❷ Adding a new attribute **cd2** to the TDataServer with a mathematical formula, precising its title and unit:

$$cd2 \;=\; \sqrt{x_2 * x_1}$$

❸ Plots the scatterplot of the variable **cd2** versus the variable **cd1**

The obtained graph is:

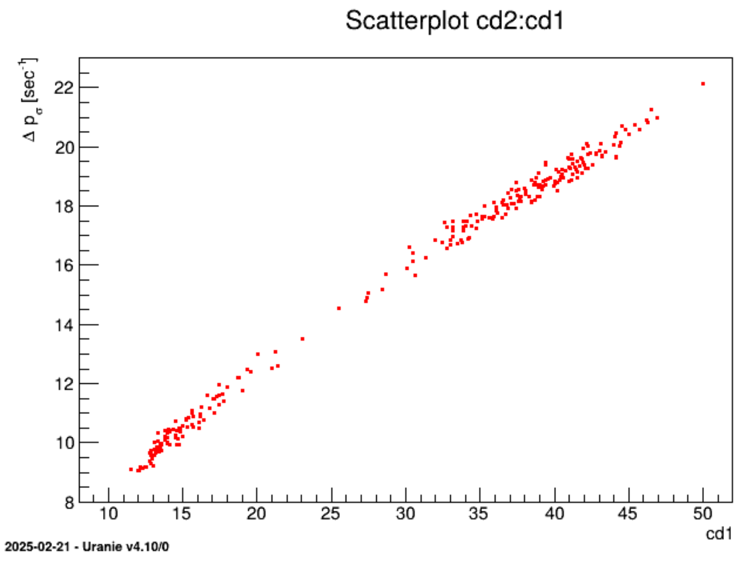


Figure II.39: Scatterplot of added attributes

This operation is available with vector-type attribute as well. The results depends on the nature of the attributes involved in the formula, their content and the nature of the operation. As an example, a simple dataserver is created from the dummy file `tdstest.dat`:

```
#COLUMN_NAMES: x| y| a| v
#COLUMN_TYPES: V|V|D|V

1,2,3 4,5,6 2 1,2,3
7,8,9 1,2   4 4,5,6
1,4,8 2,5,4 5 7,8,9
```

It contains two vectors whose size are not constant and a double. The four usual operations have been performed (addition, subtraction, multiplication and division) using the double and a vector but also using the two vectors. The code is shown here:

```
{
  TDataServer *tdsop =new TDataServer("foo","poet");
  tdsop->fileDataRead("tdstest.dat");

  tdsop->addAttribute("x*y","x*y"); // Multiply two vectors
  tdsop->addAttribute("xovy","x/y"); // Divide two vectors
  tdsop->addAttribute("x-y","x-y"); // Subtract two vectors
  tdsop->addAttribute("x+y","x+y"); // Add two vectors

  tdsop->addAttribute("x*a","x*a"); // Multiply a vector and a double
  tdsop->addAttribute("xova","x/a"); // Divide a vector and a double
  tdsop->addAttribute("x-a","x-a"); // Subtract a vector and a double
  tdsop->addAttribute("x+a","x+a"); // Add a vector and a double

  tdsop->scan("x:y:a:x*y:xovy:x+y:x-y:x*a:xova:x+a:x-a","","colsize=3 col=1:1:1::4::::4:");
  }
```

and it gives as a results:

```
*************************************************************************************
*    Row   * Instance * x * y * a * x*y * xovy * x+y * x-y * x*a * xova * x+a * x-a *
*************************************************************************************
*        0 *        0 * 1 * 4 * 2 *   4 * 0.25 *   5 *  -3 *   2 *  0.5 *   3 *  -1 *
*        0 *        1 * 2 * 5 * 2 *  10 *  0.4 *   7 *  -3 *   4 *    1 *   4 *   0 *
*        0 *        2 * 3 * 6 * 2 *  18 *  0.5 *   9 *  -3 *   6 *  1.5 *   5 *   1 *
*        1 *        0 * 7 * 1 * 4 *   7 *    7 *   8 *   6 *  28 * 1.75 *  11 *   3 *
*        1 *        1 * 8 * 2 * 4 *  16 *    4 *  10 *   6 *  32 *    2 *  12 *   4 *
*        1 *        2 * 9 *   * 4 *   0 *      *   0 *   0 *  36 * 2.25 *  13 *   5 *
*        2 *        0 * 1 * 2 * 5 *   2 *  0.5 *   3 *  -1 *   5 *  0.2 *   6 *  -4 *
*        2 *        1 * 4 * 5 * 5 *  20 *  0.8 *   9 *  -1 *  20 *  0.8 *   9 *  -1 *
*        2 *        2 * 8 * 4 * 5 *  32 *    2 *  12 *   4 *  40 *  1.6 *  13 *   3 *
*************************************************************************************
```

Summary: Adding attributes

Two kinds of method, allow to add an attribute to a `TDataServer`:

1. With attribute properties: `addAttribute` (**TAttribute** *att) / `addAttribute` (**TString** name)

   The pointer `att` is either of `TAttribute` type or a derived class `TAttributeFormula` or `TStochasticAttribute` type.

2. By means of other existing attributes: `addAttribute` (**TString** name, **TString** formula, **TString** label="", **TString** unity="")

   Adding an attribute by specifying its name and its mathematical formula (these two arguments are mandatory),and its title and its unit (optional arguments).

   A warning has been added if formula is requested using a string-type branch. In the case of vector, the behaviour should depend on the nature of the branches in the formula.

II.3.5 Merging two DataServer

Warning
This section is discussing the merging of two `TDataServer` not their concatenation. The first operation consists in adding new attributes from an existing `TDataServer` into another existing one, while the seconds consists in adding the content of two `TTree` object with the exact same structure. For the merging operation, a specific method `TDataServer::merge` has been written, while for the concatenation, the interested user is invited to look at the `TChain::Merge` method from ROOT.

It is sometimes necessary to merge two **`TDataServer`** to form a single one. Since the merging is done line by line, one has to check that both objects contain the same number of patterns. In Uranie-version older than 3.10.0 it was assumed that the patterns were **exactly** stored in the same order. Now the method is looking for the iterator of both `TDataServer` objects and it checks that both iterators contain the same value all along (not necessary in the same order, for instance when dealing with distributed computations). If the iterators are not found or if some iterator's values are found in one iterator but not the other (possible in some rare cases such as `OAT` sampling), the merging is done line-by-line and a warning is displayed.

This operation is common when you want to build a surface response between output variables Y and predictors X and these data are located in two different files.

Warning
The 2 objects must have the same number of patterns.

Let's take a simple example. Assuming that we have 2 `TDataServer` **tds1** and **tds2** respectively located in the ASCII files `tds1.dat` and `tds2.dat`. Another example is also provided in Section XIV.2.2.

Data file `tds1.dat`

```
#COLUMN_NAMES: x| dy| z| theta
#COLUMN_TITLES: x_{n}| "#delta y"| ""| #theta
#COLUMN_UNITS: N| Sec| KM/Sec| M^{2}

1 1 11 11
1 2 12 21
1 3 13 31
2 1 21 12
2 2 22 22
2 3 23 32
3 1 31 13
3 2 32 23
3 3 33 33
```

Data file `tds2.dat`

```
#COLUMN_NAMES: x2| y| u| ua

1 1 102 11
1 2 104 12
1 3 106 13
2 1 202 21
2 2 204 22
2 3 206 23
3 1 302 31
3 2 304 32
3 3 306 33
```

Both `TDataServer` incorporate 9 patterns.

These 2 *ASCII* files must be loaded in 2 `TDataServer` (cf Section II.3.2), the merging being done by calling the `merge` method of the first `TDataServer`

```
{
  TDataServer * tds1 = new TDataServer();
  tds1->fileDataRead("tds1.dat");
  TDataServer * tds2 = new TDataServer();
  tds2->fileDataRead("tds2.dat");

  tds1->merge(tds2);
  }
```

Thus, the object `tds1` also contains the attributes of the second `TDataServer` `tds2`

```
**************************************************************
*    Row   * tds * x. * dy * z. * theta * x2 * y. * u.u * ua *
**************************************************************
*        0 *   1 *  1 *  1 * 11 *    11 *  1 *  1 * 102 * 11 *
*        1 *   2 *  1 *  2 * 12 *    21 *  1 *  2 * 104 * 12 *
*        2 *   3 *  1 *  3 * 13 *    31 *  1 *  3 * 106 * 13 *
*        3 *   4 *  2 *  1 * 21 *    12 *  2 *  1 * 202 * 21 *
*        4 *   5 *  2 *  2 * 22 *    22 *  2 *  2 * 204 * 22 *
*        5 *   6 *  2 *  3 * 23 *    32 *  2 *  3 * 206 * 23 *
*        6 *   7 *  3 *  1 * 31 *    13 *  3 *  1 * 302 * 31 *
*        7 *   8 *  3 *  2 * 32 *    23 *  3 *  2 * 304 * 32 *
*        8 *   9 *  3 *  3 * 33 *    33 *  3 *  3 * 306 * 33 *
**************************************************************
```

Summary: Merging two `TDataServer`

- `merge` (`TDataServer` *tds2, const char* varexpinput="*")

  Adds the attributes of the `TDataServer` *tds2* to the current `TDataServer`. The method is checking for iterators and their content to perform the merging, as already stated above.

  If the second argument is precised, only the requested attributes of *tds2* are added to the correct tds.

II.3.6 Pattern selection

It can be necessary during a study to apply filters on the patterns; i.e. to include or to exclude patterns depending on criterion. For example, to select the patterns with x1 lower than 3.0 and x2 lower than 55.0 from the `geyser` database, Uranie code is as follows:

```
tdsGeyser->setSelect("( x1<3.0 ) && ( x2<55.)");
tdsGeyser->draw("x2:x1");
```

The obtained figure is:

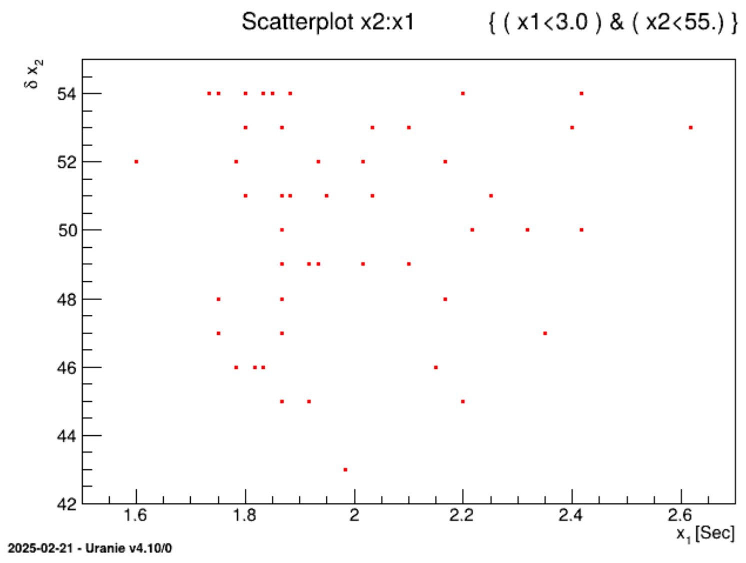


Figure II.40: Graph with a selection definition

The result of the `scan` method applied on this `TDataServer` object yields:

```
************************************************
*    Row   *        x1 *        x2 * n__iter__ *
************************************************
*        1 *       1.8 *        54 *         2 *
*        8 *      1.95 *        51 *         9 *
*       10 *     1.833 *        54 *        11 *
*       13 *      1.75 *        47 *        14 *
*       15 *     2.167 *        52 *        16 *
...
*      268 *      2.15 *        46 *       269 *
*      270 *     1.817 *        46 *       271 *
************************************************
==> 53 selected entries
```

We have obtained 53 patterns among 278 respecting the given criterion without having to specify this criterion for the `draw` and `scan` calls. To get the same result, we could have executed the following command as well:

```
tdsGeyser->draw("x2:x1", "( x1<3.0 ) && ( x2<55.)");
//tdsGeyser->scan("*", "( x1<3.0 )  && ( x2<55.)");
```

However, in this case, we have to repeat the criterion for each command.

It is also possible to exclude patterns coming from `TDataServer` with the `setCut` method.

```
tdsGeyser->setCut(TString("x1 >= 3."));
tdsGeyser->draw("x2:x1");
```

The obtained figure is as follows:

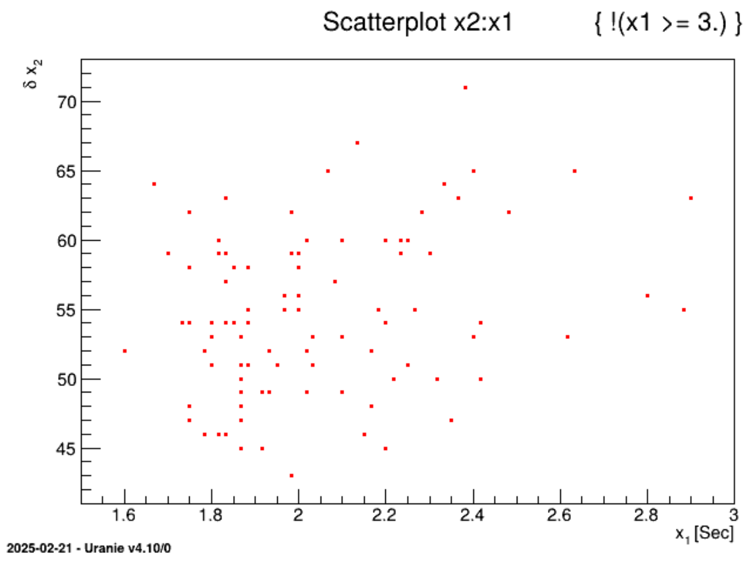


Figure II.41: Graph with a definition of *Cut*

It can be noticed in the title of Figure II.41, that it simply corresponds to the opposite of criterion's meaning with respect to the one given by the `setCut` method with the **!()** character. Thus, the `setCut` method consists in passing on the negation of the criterion to the `setSelect` command.

Finally, it is perfectly possible to delete the current filters with the methods `clearSelect` and `clearCut`, and retrieving the unfiltered results.

Tip

Every modifications of the ongoing selection (meaning doing a new selection or removing it) is now clearing automatically the vectors that contain statistical properties of attributes and the database of already computed quantiles. This is reminded with an information line shown below:

```
<URANIE::INFO>  Selection is changing ==> clearing the TAttribute computed ↩
    statistics and quantiles
```

Summary: Filters management regarding patterns

- `setSelect` (**TString** sselect)

  Patterns selection ensuring the criterion *sselect*

- `setCut` (**TString** scut)

  Excluding the patterns ensuring the criterion *scut*

- `clearSelect`() / `clearCut`()

  Deletes the current filter.

Any modification of the selection is now clearing the vector of statistic (mean, std, min and max) introduced in Section II.2.2 as well as the map containing quantiles. This is done in order to be sure that the method used to performed these estimation are called once again to take into account the change in selection. An informative message is displayed to remind this clearing.

II.3.7 Export to an ASCII file

In the same way as the data are imported from an ASCII file, we can also save the data of a `TDataServer` in an ASCII file. Currently, four methods of export are available in Uranie:

- using the same format as that observed during import ("Salome Table");
- using a **C** file containing the data vectors that can be inserted in a **C** program.
- using the **NeMo** format: the generated file is useful for the **NeMo** tool for constructing neural response surface developed at STMF.
- using the **JSON** format: the generated file is easily transferable to any other program that include the JSON protocol. This file can also be read through the `fileDataReadJSon` method of a `TDataServer` object.

```
TDataServer * tdsGeyser = new TDataServer("tdsgeyser", "geyser database");
tdsGeyser->fileDataRead("geyser.dat");
tdsGeyser->addAttribute("y", "sqrt(x2) * x1");

tdsGeyser->exportData("newfile.dat"); ❶[1]
tdsGeyser->exportDataHeader("newfile.C", "x1:x2:y"); ❷[1]
tdsGeyser->exportDataNeMo("newfile.nemo", "x1:x2", "y", "x2<75.0"); ❸[1]
tdsGeyser->exportDataJSon("newfile.json"); ❹[1]
```

Data export from a `TDataServer` in an ASCII file

❶ Export the data of the `TDataServer` *tdsGeyser* in an ASCII file `"newfile.dat"`:

```
#NAME: tdsgeyser
#TITLE: Database of the geyser
#DATE: Tue Oct  9 15:41:29 2007
#COLUMN_NAMES: x1| x2| y| n__iter__

3.600000000e+00 7.900000000e+01 3.199749990e+01 1
1.800000000e+00 5.400000000e+01 1.322724461e+01 2
```

```
3.333000000e+00 7.400000000e+01 2.867155012e+01 3
2.283000000e+00 6.200000000e+01 1.797635998e+01 4
4.533000000e+00 8.500000000e+01 4.179219502e+01 5
...
2.150000000e+00 4.600000000e+01 1.458200946e+01 269
4.417000000e+00 9.000000000e+01 4.190334127e+01 270
1.817000000e+00 4.600000000e+01 1.232349358e+01 271
4.467000000e+00 7.400000000e+01 3.842658697e+01 272
```

❷ Exports the data of attributes *x1, x2 and y* of the `TDataServer` *tdsGeyser* in the ASCII file `"newfile.C"` A format "Header".

```
// File "newfile.C" generated by ROOT v5.17/04
// DateTime Tue Oct  9 15:41:30 2007
// DataServer: Name="tdsgeyser" Title="Database of the geyser" Select=""

#define essai_nPattern 272

//  Attribute Name="x1" Title=" x_{1}"
Double_t x1[essai_nPattern] = {
3.600000000e+00,
1.800000000e+00,
...
1.817000000e+00,
4.467000000e+00,
};
// End of attribute x1

//  Attribute Name="x2" Title=" #delta x_{2}"
Double_t x2[essai_nPattern] = {
7.900000000e+01,
5.400000000e+01,
...
1.232349358e+01,
3.842658697e+01,
};
// End of attribute y

// End of File newfile.C
```

❸ Exports the data of the `TDataServer` *tdsGeyser* in an ASCII file `newfile.nemo` with format *NeMo* with *x1:x2* as input vector, *"y"* as output and applying a filter "x2<75.0"

```
#NombreExemples 126
#NombreEntrees 2
#NombreSorties 1

1.800000000e+00 5.400000000e+01 1.322724461e+01
3.333000000e+00 7.400000000e+01 2.867155012e+01
2.283000000e+00 6.200000000e+01 1.797635998e+01
2.883000000e+00 5.500000000e+01 2.138090024e+01
...
2.150000000e+00 4.600000000e+01 1.458200946e+01
1.817000000e+00 4.600000000e+01 1.232349358e+01
4.467000000e+00 7.400000000e+01 3.842658697e+01
```

❹ Exports the data of the `TDataServer` *tdsGeyser* in an ASCII file `newfile.json` with format *JSON*.

```
{
  "_metadata" :
```

```
  {
    "_comment" : "CurrentComment",
    "date" : "Fri Oct 28 10:41:44 2016",
    "short_names" : [ "x1", "x2", "geyser__n__iter__" ],
    "table_description" : "Les donnees du geyser",
    "table_name" : "geyser",
    "types" : [ "D", "D", "D" ],
    "units" : [ "Sec", "", "" ]
  },
  "items" :
  [

    {
      "geyser__n__iter__" : 1.0,
      "x1" : 3.60,
      "x2" : 79.0
    },

    {
      "geyser__n__iter__" : 2.0,
      "x1" : 1.80,
      "x2" : 54.0
    },



    {
      "geyser__n__iter__" : 272.0,
      "x1" : 4.4670,
      "x2" : 74.0
    }
  ]
}
```

Summary: Exportation of a `TDataServer` to ASCII format

- `exportData` (**const char*** filename, **const char** * varexp="*", **const char** * select="")

  Exportation of the data of attributes *"varexp"* of the `TDataServer` in the file *"filename"* in ASCII format, **"Salome table"** type, applying the filter contained in *"select"*. The filter "select" is added to the permanent selection (*c.f.* Section II.3.6) of the `TDataServer`.

- `exportDataJSon` (**const char*** filename, **const char** * varexp="*", **const char** * select="")

  Exportation of the data of attributes *"varexp"* of the `TDataServer` in the file *"filename"* in JSON format, applying the filter contained in *"select"*. The filter "select" is added to the permanent selection (*c.f.* Section II.3.6) of the `TDataServer`.

- `exportDataHeader` (**const char*** filename, **const char** * varexp="*", **const char** * select="")

  Exportation of the data of attributes *"varexp"* of the `TDataServer` in the file *"filename"* in ASCII format for the use of a *C/C++* program. The "select" filter is added to the permanent selection (*c.f.* Section II.3.6) of the `TDataServer`.

- `exportDataNeMo` (**const char*** filename, **const char** * varexpinput, **const char** * varexpoutput, **const char** * select="")

  Exportation of the `TDataServer` data din the file *"filename"* `TDataServer` in ASCII format **NeMo** with attributes *"varexpinput"* as inputs (separated by the character ":") and attributes *"varexpoutput"* as outputs (separated by the character ":") and applying the filter contained in option. The "select" filter is added to the the permanent selection (*c.f.* Section II.3.6) of the `TDataServer`

II.4 Statistical treatments and operations

This section presents the statistical computations allowed on the attributes in a `TDataServer` through the following five main groups of method.

All the methods have been adapted to cope with the case of vector: a check is performed, allowing the computation to be done only if the number of element in the vector is constant (at least throughout the selection if one is requested). If the constant-size criterion is not fulfilled, the considered vector is disregarded for this method. The methods detailed here are:

- The normalisation of variable, in Section II.4.1
- The ranking of variable, in Section II.4.2
- The elementary statistic computation, in Section II.4.3
- The quantile estimation, in Section II.4.4
- The correlation matrix determination, in Section II.4.5

II.4.1 Normalising the variable

The normalisation function `normalize` can be called to create new attributes whose range and dispersion depend on the chosen normalisation method. This function can be called without argument but also using up to four ones (the list of which is given in the summary below). Up to now, there are four different ways to perform this normalisation:

- centered-reducted (enum value **kCR**): the new variable values are computed as $\tilde{x} = \frac{x - \mu_x}{\sigma_x}$
- centered (enum value **kCentered**): the new variable values are computed as $\tilde{x} = x - \mu_x$
- reduced to $[-1,1]$ (enum value **kMinusOneOne**): the new variable values are computed as $\tilde{x} = 2.0 \times \frac{x - x_{\text{Min}}}{x_{\text{Max}} - x_{\text{Min}}} - 1.0$
- reduced to $[0,1]$ (enum value **kZeroOne**): the new variable values are computed as $\tilde{x} = \frac{x - x_{\text{Min}}}{x_{\text{Max}} - x_{\text{Min}}}$

The following piece of code shows how to use this function on a very simple dataserver, focusing on a vector whose values goes from 1 to 9 over three events.

```
{
  TDataServer *tdsop =new TDataServer("foo","pouet");
  tdsop->fileDataRead("tdstest.dat");

  //Compute a global normalisation of v, CenterReduced
  tdsop->normalize("v","GCR",TDataServer::kCR,true);
  //Compute a normalisation of v, CenterReduced (not global but entry by entry)
  tdsop->normalize("v","CR",TDataServer::kCR,false);

  //Compute a global normalisation of v, Centered
  tdsop->normalize("v","GCent",TDataServer::kCentered);
  //Compute a normalisation of v, Centered  (not global but entry by entry)
  tdsop->normalize("v","Cent",TDataServer::kCentered,false);

  //Compute a global normalisation of v, ZeroOne
  tdsop->normalize("v","GZO",TDataServer::kZeroOne);
  //Compute a normalisation of v, ZeroOne (not global but entry by entry)
  tdsop->normalize("v","ZO",TDataServer::kZeroOne,false);

  //Compute a global normalisation of v, MinusOneOne
  tdsop->normalize("v","GMOO",TDataServer::kMinusOneOne,true);
  //Compute a normalisation of v, MinusOneOne (not global but entry by entry)
  tdsop->normalize("v","MOO",TDataServer::kMinusOneOne,false);

  tdsop->scan("v:vGCR:vCR:vGCent:vCent:vGZO:vZO:vGMOO:vMOO","","colsize=4 col=2:5:::::::") ↩
      ;
  }
```

The normalisation is performed using all methods, first with the global flag set to true (the suffix always starts with "G" for global) and then with the more local approach. The result of the `scan` method is given below:

```
**************************************************************************************
*    Row   * Instance *  v *  vGCR *  vCR * vGCe * vCen * vGZO *  vZO * vGMO * vMOO *
**************************************************************************************
*        0 *        0 *  1 * -1.46 *   -1 *   -4 *   -3 *    0 *    0 *   -1 *   -1 *
*        0 *        1 *  2 * -1.09 *   -1 *   -3 *   -3 * 0.12 *    0 * -0.7 *   -1 *
*        0 *        2 *  3 * -0.73 *   -1 *   -2 *   -3 * 0.25 *    0 * -0.5 *   -1 *
*        1 *        0 *  4 * -0.36 *    0 *   -1 *    0 * 0.37 *  0.5 * -0.2 *    0 *
*        1 *        1 *  5 *     0 *    0 *    0 *    0 *  0.5 *  0.5 *    0 *    0 *
*        1 *        2 *  6 * 0.365 *    0 *    1 *    0 * 0.62 *  0.5 * 0.25 *    0 *
*        2 *        0 *  7 * 0.730 *    1 *    2 *    3 * 0.75 *    1 *  0.5 *    1 *
*        2 *        1 *  8 * 1.095 *    1 *    3 *    3 * 0.87 *    1 * 0.75 *    1 *
*        2 *        2 *  9 * 1.460 *    1 *    4 *    3 *    1 *    1 *    1 *    1 *
**************************************************************************************
```

Summary: normalize

The method is `normalize(const char* varexp="", const char* suffix="_CR", ENormalisation method=kCR, bool global=true)` and is adapted to deal with constant-size vectors. It creates a new attribute for every attribute concerned by the call and can be called with 0 to 4 arguments;

- `const char* varexp=""`: the first argument is the list of attributes on which the normalisation is applied. Left as it is, all attributes will be read and transformed in a new set of attributes.
- `const char* suffix="_CR"`: the second argument describes the suffix that will be added to the attribute name to obtain the new normalised attribute name.
- `ENormalisation method=kCR`: the third argument is an enumerator that describes the method chosen to perform the normalisation (the list of which is provided above)
- `bool global=true` : the fourth argument is only useful in the case where the attribute is a vector. In this case one can consider two ways of normalising the entries (despite the chosen method): either normalise every iteration of the constant size vector with respect to the same iteration in other events, without considering the entirety of the vector (meaning the other iterations of the vector), or normalise the entries considering that all entries of a vector are a part of a global pull of number which can be described by one mean and standard deviation. The former case corresponds to global equal to false while the latter is the opposite (and default). This is possible thanks to the modification done on the method performing the statistical treatment

II.4.2 Computing the ranking

The ranking of variable is used in many methods that are focusing more on monotony than on linearity (this is discussed throughout this documentation when coping with regression, correlation matrix, ...). The way this is done in Uranie is the following: for every attribute considered, (which means all attributes by default if the function is called without argument) a new attribute is created, whose name is constructed as the name of the considered attribute with the prefix "Rk_". The ranking consists, for a simple double-precision attribute, in assigning to each attribute entry an integer, that goes from 1 to the number of patterns, following an order relation (in Uranie it is chosen so that 1 is the smallest value and N is the largest one).

This method has been modified in order to cope with constant size vectors, but also to stabilise its behaviour when going from one compiler version to another. The first modification only consists in considering every element of a constant-size vector independent from the others, so every element is in fact treated as if they were different attributes. The second part is more technical as the sorting method has been changed to use the `std::stable_sort` insuring that platforms (operating systems and compiler versions) will have the same behaviour. The main problem was raising when two patterns had the same value for the attribute under study. In this case, the ranking was not done in the same way depending on the version of the compiler. Now it should be treated in the same way: if two or more patterns have the same value for a specific attribute, the first met in the array of attribute value will have the value i while the second one will be affected with $i+1$ and so on... Here is a small example of this computation:

```
{
  TDataServer *tdsGeyser =new TDataServer("geyser","poet");
  tdsGeyser->fileDataRead("geyser.dat");
  tdsGeyser->computeRank("x1");
  tdsGeyser->computeStatistic("Rk_x1");

  cout<<"NPatterns="<<tdsGeyser->getNPatterns()<<";  min(Rk_x1)= "<<tdsGeyser->getAttribute ↩
      ("Rk_x1")->getMinimum()
      <<";  max(Rk_x1)= "<<tdsGeyser->getAttribute("Rk_x1")->getMaximum()<<endl;
  }
```

This macro should returns

```
NPatterns=272;  min(Rk_x1)= 1;  max(Rk_x1)= 272
```

Summary: computeRank

- `computeRank`(**const char*** varexp="*", **option*** option)

  Create a new attribute for every attribute requested (or for all attributes if no argument is provided)

  String-type and non-constant-vector-type attribute are disregarded and a warning is shown to let the user know.

II.4.3 Computing the elementary statistic

The `TDataServer` provides a method to determine the four simplest statistical notions: the minimum, maximum, average and standard deviation. It can be simply called without argument (running then over all the attributes), or with a restricted list of attributes. A second possible argument is a selection criteria (which is not applied through the `setSelect` method so not changing the behaviour of the `TDataServer` in the other method).

```
{
  TDataServer *tdsGeyser =new TDataServer("geyser","poet");
  tdsGeyser->fileDataRead("geyser.dat");
  tdsGeyser->computeStatistic("x1");

  cout<<"min(x1)= "<<tdsGeyser->getAttribute("x1")->getMinimum()<<";  max(x1)= "<<tdsGeyser ↩
      ->getAttribute("x1")->getMaximum()
      <<";  mean(x1)= "<<tdsGeyser->getAttribute("x1")->getMean()<<";  std(x1)= "<< ↩
          tdsGeyser->getAttribute("x1")->getStd()<<endl;
  }
```

It returns the following line

```
min(x1)= 1.6;  max(x1)= 5.1;  mean(x1)= 3.48778;  std(x1)= 1.14137
```

II.4.3.1 Specific case of vectors

As stated in Section II.2.2, these information are now stored in vectors because of the new possible attribute nature. In the case of constant-size vectors whose dimension is N, the attribute-statistical vectors are filled with the statistical information considering every elements of the input vector independent from the others. This results in attribute-statistical vectors of size $N+1$, the extra element being the statistical information computed over the complete vector. Here is an example of `computeStatistic` use with the *tdstest.dat* file already shown in Section II.3.4:

```
{
  TDataServer *tdsop =new TDataServer("foo","poet");
  tdsop->fileDataRead("tdstest.dat");

  //Considering every element of a vector independent from the others
  tdsop->computeStatistic("x");
  TAttribute *px = tdsop->getAttribute("x");

  cout<<"min(x[0])= "<<px->getMinimum(0)<<";  max(x[0])= "<<px->getMaximum(0)
      <<";  mean(x[0])= "<<px->getMean(0)<<";  std(x[0])= "<<px->getStd(0)<<endl;
  cout<<"min(x[1])= "<<px->getMinimum(1)<<";  max(x[1])= "<<px->getMaximum(1)
      <<";  mean(x[1])= "<<px->getMean(1)<<";  std(x[1])= "<<px->getStd(1)<<endl;
  cout<<"min(x[2])= "<<px->getMinimum(2)<<";  max(x[2])= "<<px->getMaximum(2)
      <<";  mean(x[2])= "<<px->getMean(2)<<";  std(x[2])= "<<px->getStd(2)<<endl;
  cout<<"min(xtot)= "<<px->getMinimum(3)<<";  max(xtot)= "<<px->getMaximum(3)
      <<";  mean(xtot)= "<<px->getMean(3)<<";  std(xtot)= "<<px->getStd(3)<<endl;

  //Statistic for a single realisation of a vector, not considering other events
  tdsop->addAttribute("Min_x","Min$(x)");
  tdsop->addAttribute("Max_x","Max$(x)");
  tdsop->addAttribute("Mean_x","Sum$(x)/Length$(x)");

  tdsop->scan("x:Min_x:Max_x:Mean_x","","colsize=5 col=2::::");
  }
```

The first computation is filling the vector of statistical elementary in the concerned attribute x. The first, second and third `cout` line in the previous piece of code are dumping the statistical characteristics respectively for the first, second and third element of the vector. The fourth one is giving the main characteristics considering the complete vector and all the entries. The results of this example are shown below:

```
min(x[0])= 1;  max(x[0])= 7;  mean(x[0])= 3;  std(x[0])= 3.4641
min(x[1])= 2;  max(x[1])= 8;  mean(x[1])= 4.66667;  std(x[1])= 3.05505
min(x[2])= 3;  max(x[2])= 9;  mean(x[2])= 6.66667;  std(x[2])= 3.21455
min(xtot)= 1;  max(xtot)= 9;  mean(xtot)= 4.77778;  std(xtot)= 3.23179
```

This implementation has been chosen as ROOT offers access to another way of computing these notions if one wants to consider every element of a vector, assuming that every event is now independent from the others. Indeed it is possible to get the minimum, maximum and mean of a vector on an event-by-event basis by introducing a new attribute with a formula, as done in Section II.3.4. This is the second part of the code shown in the box above (using specific function from ROOT, that needs the sign "$" to be recognised). The results are shown below:

```
*****************************************************
*    Row   * Instance *  x * Min_x * Max_x * Mean_x *
*****************************************************
*        0 *        0 *  1 *     1 *     3 *      2 *
*        0 *        1 *  2 *     1 *     3 *      2 *
*        0 *        2 *  3 *     1 *     3 *      2 *
*        1 *        0 *  7 *     7 *     9 *      8 *
*        1 *        1 *  8 *     7 *     9 *      8 *
*        1 *        2 *  9 *     7 *     9 *      8 *
*        2 *        0 *  1 *     1 *     8 * 4.3333 *
*        2 *        1 *  4 *     1 *     8 * 4.3333 *
*        2 *        2 *  8 *     1 *     8 * 4.3333 *
*****************************************************
```

Summary: computeStatistic

- `computeStatistic`(**const char***varexp = "*", **const char***selection = "", **Option_t** *option = "")

  Estimate Mean, Std, Maximum and Minimum for attributes requested in varexp, with an optional additional selection. If no argument is provided, a loop over all attributes is performed.

- `getMean(int iel=0)`, `getStd(int iel=0)`, `getMinimum(int iel=0)`, `getMaximum(int iel=0)`

  Method from `TAttribute` class, made to give access to the computed statistic information. The argument is the element number in the corresponding vector, the default being one. The size of the vector is larger than the original vector because of the statistic computation is also performed on all elements at once. The size of this vector is provided by these functions:

- `getMeanSize()`, `getStdSize()`, `getMinimumSize()`, `getMaximumSize()`

  Method from `TAttribute` class, providing access to the size of the statistical-vectors.

- The statistical vectors are cleared as soon as the overall selection is modified.

II.4.4 The quantile computation

There are several ways of estimating the quantiles implemented in Uranie. This part describes the most commonly used and starts with a definition of quantile.

A quantile x_p, as discussed in the following parts, for p a probability going from 0 to 1, is the lowest value of the random variable X leading to $P\{X \leq x_p\} = p$. This definition holds equally if one is dealing with a given probability distribution (leading to a theoretical quantile), or a sample, drawn from a known probability distribution or not (leading to an empirical quantile). In the latter case, the sample is split into two sub-samples: one containing pN points, the other one containing $(1-p)N$ points. It can be easily pictured by looking at Figure II.44 which represents the cumulative distribution function (CDF) of the attribute x_2. The quantile at 50 percent for x_2 can be seen by drawing an horizontal line at 0.5, the value of interest being the one on the abscissa where this line crosses the CDF curve.

II.4.4.1 computeQuantile

For a given probability p, the corresponding quantile q is given by:

$$q = (1-p)x_k + px_{k+1}$$

where x_k is the k-Th smallest value of the attribute set-of-value (whose size is N). The way k is computed is discussed later on, as a parameter of the functions.

The implementation and principle has slightly changed in order to be able to cope with vectors (even though the previous logic has been kept for consistency and backward compatibility). Let's start with an example of the way it was done with the two main methods whose name are the same but differ by their signature.

```
double aproba=0.5, aquant=0;
tdsGeyser->computeQuantile("x2", aproba, aquant); ❶[1]

double Proba[2]={0.05,0.95}; double Quant[2]={0,0}; ❷[2]
tdsGeyser->computeQuantile("x2", 2, Proba, Quant);

cout<<"Quant[0]="<<Quant[0]<<"; aquant="<<aquant<<"; Quant[1]="<<Quant[1]<<endl; ❸[3]
```

Description of the methods and results

❶ This function takes here three mandatory arguments: the attribute name, the value of the chosen probability and a double whose value will be changed in the function to the estimated result.

❷ This function takes here four mandatory arguments: the attribute name, the number N_q of calculation to be done, the values of the chosen probability transmitted as an array of size N_q and another array of size N_q whose value will be changed in the function to the estimated results.

❸ This line shows the results of the three previous computations.

This implementation has been slightly modified for two reasons: to adapt the method to the case of vectors and to store easily the results and prevent from recomputing already existing results. Even though the previous behaviour is still correct, the information is now stored in the attribute itself, as a vector of map. For every element of a vector, a map of format `map<double,double>` is created: the first double is the key, meaning the value of probability provided by the user, while the second double is the results. It is now highly recommended to use the method of the `TAttribute`, that gives access to these maps for two reasons: the results provided by the methods detailed previously are only correct for the last element of a vector, and the vector of map just discussed here is cleared as soon as the general selection is modified (as for the elementary statistical-vectors discussed in Section II.4.3). The next example uses the following input file, named `aTDSWithVectors.dat`:

```
#NAME: cho
#COLUMN_NAMES: x|rank
#COLUMN_TYPES: D|V

0 0,1
1 2,3
2 4,5
3 6,7
4 8,9
```

From this file, the following code (that can be find in Section XIV.2.14) shows the different methods created in the attribute class in order for the user to get back the computed values:

```
{
    TDataServer *tdsvec = new TDataServer("foo", "bar");
    tdsvec->fileDataRead("aTDSWithVectors.dat");

    double probas[3]={0.2, 0.6, 0.8}; double quants[3];
    tdsvec->computeQuantile("rank", 3, probas, quants);

    TAttribute *prank = tdsvec->getAttribute("rank");
    int nbquant;
    prank->getQuantilesSize(nbquant); // (1)
    cout << "nbquant = " << nbquant << endl;

    double aproba=0.8; double aquant;
    prank->getQuantile(aproba, aquant); // (2)
    cout << "aproba = " << aproba << ", aquant = " <<
    aquant << endl;

    double theproba[3], thequant[3];
    prank->getQuantiles(theproba, thequant); // (3)
    for(int i_quant=0; i_quant<nbquant; ++i_quant) {
        cout << "(theproba, thequant)[" << i_quant << "] = "
        << "(" << theproba[i_quant] << ", " <<
        thequant[i_quant] << ")" << endl;
    }
    vector<double> allquant;
    prank->getQuantileVector(aproba, allquant); // (4)
    cout << "aproba = " << aproba << ", allquant = ";
    for(double quant_i: allquant)
        cout << quant_i << " ";
    cout << endl;
}
```

Description of the methods and results

(1) This method changes the value of nbquant to the number of already computed and stored values of quantiles. A second argument can be provided to state which element of the vector is concerned (if the attribute under study is a vector, the default value being 0).

(2) This method changes the value of aquant to the quantile value corresponding to a given probability aproba. A second argument can be provided to state which element of the vector is concerned (if the attribute under study is a vector, the default value being 0).

(3) As previously, this method changes the values of thequant to the quantile values corresponding to given probabilities stores in theproba. A second argument can be provided to state which element of the vector is concerned (if the attribute under study is a vector, the default value being 0). Warning: the size of both arrays has to be carefully set. It is recommended to use the `getQuantilesSize` method ahead of this one.

(4) This method fills the provided vector allquant with the quantile value of all element of the attribute under study corresponding to a given probability aproba.

The results of this example are shown below:

```
nbquant = 3
aproba = 0.8, aquant = 6.4
(theproba, thequant)[0] = (0.2, 1.6)
(theproba, thequant)[1] = (0.6, 4.8)
(theproba, thequant)[2] = (0.8, 6.4)
aproba = 0.8, allquant = 6.4 7.4
```

Summary: computeQuantile

- `computeQuantile`(**const char** *attName, **Double_t** proba, **Double_t** &quantile, **Int_t** type = 7);
- `computeQuantile`(**const char** *attName, **Int_t** nProba, **Double_t** * proba, **Double_t** *quantile, **Int_t** type = 7);

  The methods are discussed above. The last parameter determines how k is computed. For discontinuous cases:

  - 1: $k = \lfloor p \times N \rfloor$; if $p \times N = k$, $q = x_k$. $q = x_{k+1}$ otherwise.
  - 2: $k = \lfloor p \times N \rfloor$; if $p \times N = k$, $q = 1/2 \times (x_k + x_{k+1})$. $q = x_{k+1}$ otherwise.
  - 3: $k = \lfloor p \times N - 0.5 \rfloor$; if $p \times N - 0.5 = k$ and k is even, $q = x_k$. $q = x_{k+1}$ otherwise., default in SAS.

  For piece-wise linear interpolations:

  - 4: $k = \lfloor p \times N \rfloor$
  - 5: $k = \lfloor p \times N - 0.5 \rfloor$
  - 6: $k = \lfloor p \times (N+1) \rfloor$, default in Minitab and SPSS.
  - 7: $k = \lfloor p \times (N-1) + 1 \rfloor$, default in ROOT, S and R.
  - 8: $k = \lfloor p \times (N + 1/3) + 1/3 \rfloor$, approximately median unbiased.
  - 9: $k = \lfloor p \times (N + 1/4) + 3/8 \rfloor$, approximately unbiased if x is normally distributed.

II.4.4.2 α-quantile

The α-quantile can be evaluated by several ways:

- Control variate,
- Importance sampling.

Control variate

To estimate the α-quantile by control variate, you must use the `computeQuantileCV` method. The procedure to do this estimation is the following:

- **If the control variate is determined in the macro**: A `TDataServer` is necessary and a surrogate model, like "linear regression" or "artificial neural network", needs to be built from this dataserver and exported into a file (*c.f.* Chapter V). This model will enable the creation of the control variate.

```
// Build the SR ( Linear regression + ANN)
TLinearRegression *tlin = new TLinearRegression(tds,"rw:r:tu:tl:hu:hl:l:kw", sY);
tlin->estimate();
tlin->exportFunction("c++", "_SR_rl_", "SRrl");

TANNModeler* tann=new TANNModeler(tds, Form("%s,8,%s",sinput.Data(),sY.Data()));
tann->train(3, 2, "test");
tann->setDrawProgressBar(kFALSE);
tann->exportFunction("c++", "_SR_ann_", "SRann");
```

  A variable that represents the control-variate is added to the `TDataServer`. It is built by means of the surrogate model.

```
//build Z
gROOT->LoadMacro("_SR_rl_.C");
TLauncherFunction *tlfz = new TLauncherFunction(tds, "SRrl",sinput,"Z");
tlfz->setDrawProgressBar(kFALSE);
tlfz->run();
```

The Empirical α-quantile of the control variate needs to be evaluated. You can do it with the followings commands:

```
TDataServer *tdsza = new TDataServer( Form("%s_zalpha", tds2->GetName()), "Ex. flowrate");
for(Int_t i=0; i< nattinput; i++)
tdsza->addAttribute( tds2->getAttribute(i));

TSampling *fsza = new TSampling(tdsza, "lhs", 6000);
fsza->generateSample();

TLauncherFunction * tlfrlza = new TLauncherFunction(tdsza, "SRrl", sinput, "Zrl");
tlfrlza->setDrawProgressBar(kFALSE);
tlfrlza->run();

tdsza->computeQuantile("Zrl", dAlpha, dZrla);
cout<< dZrla  << endl;
```

Then, the estimation of the α-quantile can be made by using the `computeQuantileCV` method.

```
tds->computeQuantileCV("yhat",alpha,"Z",dZrla, dY, rho);
```

Summary: computeQuantileCV

- `computeQuantileCV` (**TString** yname, **Double_t** alpha, **TString** zname, **Double_t** zapha, **Double_t &**yalpha, **Double_t &**rho)

  Estimates the α-quantile (*yalpha*) of the attribute *yname* thanks to the control variate *zname* of empirical α-quantile *zalpha*.

Importance sampling

To estimate the α-quantile by importance sampling, the method `computeThreshold` needs to be used. The procedure to make this estimation follows.

First, an object TImportanceSampling needs to be created. This object will allow the creation of a copy of the `TDataServer` where one of its attributes (sent in parameter) is replaced by a new attribute defined by its law (sent in parameter too).

```
TImportanceSampling * tis = new TImportanceSampling(tds2,"rw",new TNormalDistribution(" ↩
    rw_IS", 0.10,0.015),nS);
```

And then, this new `TDataServer` must be collected via the `getTDS` method.

```
TDataServer *tdsis = tis->getTDS();
```

A sampling needs to be generated for this new `TDataServer`:

```
TSampling * sampis = new TSampling(tdsis,"lhs",nS);
sampis->generateSample();
TLauncherFunction *tlfis = new TLauncherFunction(tdsis,"flowrateModel","*","Y_IS");
tlfis->setDrawProgressBar(kFALSE);
tlfis->run();
```

Now, the probability of an output variable exceeding threshold can be computed with the `computeThreshold` method.

```
ISproba = tis->computeThreshold("Y_IS",seuil);
```

For information, it is possible to compute the mean and standard deviation of this output variable.

```
double ISmean =  tis->computeMean("Y_IS");
double ISstd =  tis->computeStd("Y_IS");
```

Summary

- `TImportanceSampling` (**TDataServer *** tds, **TString** var, **TStochasticAttribute** var_IS)

  Build a `TDataServer`, copy of the `TDataServer` *tds* where the attribute *var* is replaced by the stochastic variable *var_IS*.

- `getTDS` ( )

  Return the new TDS built by the above constructor.

- Double_t `computeMean` (**TString** u)

  Compute the mean of the *u* variable .

- Double_t `computeStd` (**TString** u)

  Compute the standard deviation of the *u* variable .

- Double_t `computeThreshold` (**TString** u, **Double_t** val)

  Compute the probability of the *u* variable exceeding the *val* threshold.

II.4.4.3 Wilks-quantile computation

The Wilks quantile computation is an empirical estimation, based on order statistic which allows to get an estimation on the requested quantile, with a given confidence level β, independently of the nature of the law, and most of the time, requesting less estimations than a classical estimation. Going back to the empirical way discussed in Section II.4.4.1: it consists, for a 95% quantile, in running 100 computations, ordering the obtained values and taking the one at either the 95-Th or 96-Th position (see the discussion on how to choose k in Section II.4.4.1). This can be repeated several times and will result in a distribution of all the obtained quantile values peaking at the theoretical value, with a standard deviation depending on the number of computations made. As it peaks on the theoretical value, 50% of the estimation are larger than the theoretical value while the other 50% are smaller.

In the following a quantile estimation of 95% will be considered with a requested confidence level of 95% (for more details on this method, see [30]). If the sample does not exist yet, a possible solution is to estimate the minimum requested number of computations (which leads in our particular case to a sample of 59 events). Otherwise, one can ask Uranie the index of the quantile value for a given sample size, as such:

```
TDataServer *tds = new TDataServer("useless","foo");
double quant=0.95;
double CL = 0.95;
int SampleSize=200; int theindex=0;
theindex = tds->computeIndexQuantileWilks(quant, CL, SampleSize);
```

The previous lines are purely informative, and not compulsory: the method implemented in Uranie to deal with the Wilks quantile estimation will start by calling these lines and complains if the minimum numbers of points is not available. In any case, the bigger the sample is, the more accurate the estimated value is. This value is finally determined using the method:

```
tds->addAttribute( new TNormalDistribution("attribute",0,1));
TSampling * sampis = new TSampling(tds,"lhs", SampleSize);
sampis->generateSample();

double value;
tds->estimateQuantile("attribute",quant, value, CL);
```

As stated previously, this is illustrated in a use-case macro which results in Figure XIV.5. There one can see results from two classical estimations of the 95% quantile. The distribution of their results is centered around the theoretical value. The bigger the sample is, the closer the average is to the theoretical value and the smaller the standard deviation is. But in any case, there is always 50% of estimation below and 50% above the theoretical value. Looking at the Wilks estimation, one can see that only 5% and 1% of the estimations are below the theoretical value respectively for the 95% and 99% confidence level distributions (at the price of smaller sample). With a larger sample, the standard deviation of the estimated value distribution for a 95% confidence level is getting smaller.

II.4.5 Correlation matrix

The computation of the correlation matrix can be done either on the values (leading to the Pearson coefficients) or on the ranks (leading to the Spearmann coefficients). It is performed in the `computeCorrelationMatrix` method.

```
TDataServer * tdsGeyser = new TDataServer("tdsgeyser", "Geyser DataSet");
tdsGeyser->fileDataRead("geyser.dat");
tdsGeyser->addAttribute("y", "sqrt(x2) * x1");

TMatrixD matCorr = tdsGeyser->computeCorrelationMatrix("x2:x1");
cout << "Computing correlation matrix ..." << endl;
matCorr.Print();
```

```
Computing correlation matrix ...
2x2 matrix is as follows

|      0    |      1    |
-------------------------------
0 |          1      0.9008
1 |     0.9008           1
```

Same thing if computing the correlation matrix on ranks:

```
TMatrixD matCorrRank = tdsGeyser->computeCorrelationMatrix("x2:x1","", "rank");
cout << "Computing correlation matrix on ranks ..." << endl;
matCorrRank.Print();
```

```
Computing correlation matrix on ranks ...
2x2 matrix is as follows

|      0    |      1    |
-------------------------------
0 |          1      0.7778
1 |     0.7778           1
```

II.4.5.1 Special case of vector

As for all methods above, this one has been modified so that it can handle constant-size vectors (at least given the pre-selection of event from the combination of the overall selection and the one provided in the method, as a second argument). As usual, the idea is to consider all elements of a vector independent from the other. If one considers the correlation matrix computed between two attributes, one being a scalar while the other one is a constant-sized vector with 10 elements, the resulting correlation matrix will be a 11 by 11 matrix.

Here are two examples of `computeCorrelationMatrix` calls, both using the *tdstest.dat* file already shown in Section II.3.4, which contains four attributes, three of which can be used here (y being a non constant-size vector, using it in this method will bring an exception error). In the following example, two correlation matrices are computed: the first one providing the correlation of both a and x attributes while the second focus on the former and only the second element of the latter.

```
{
  TDataServer *tdsop =new TDataServer("foo","poet");
  tdsop->fileDataRead("tdstest.dat");

  // Consider a and x attributes (every element of the vector)
  TMatrixD globalOne = tdsop->computeCorrelationMatrix("x:a");
  globalOne.Print();

  // Consider a and x attributes (cherry-picking a single element of the vector)
  TMatrixD focusedOne = tdsop->computeCorrelationMatrix("x[1]:a");
  focusedOne.Print();
}
```

This should lead to the following console return, where the first correlation matrix contains all pearson correlation coefficient (considering x as a constant-size vector whose element are independent one to another) while the second on focus only on the second element of this vector (a vector's number start at 0). The following macro is shown in Section XIV.2.13.

```
4x4 matrix is as follows

     |      0    |      1    |      2    |      3    |
----------------------------------------------------------
   0 |          1       0.9449      0.6286       0.189
   1 |     0.9449            1      0.8486         0.5
   2 |     0.6286       0.8486           1      0.8825
   3 |      0.189          0.5      0.8825           1


2x2 matrix is as follows

     |      0    |      1    |
-------------------------------
   0 |          1          0.5
   1 |        0.5            1
```

Warning
When considering correlation matrix, the vectors are handled ONLY FOR PEARSON ESTIMATION. No adaptation has been made for rank ones.

Summary: Correlation matrix

- `computeCorrelationMatrix` (**const char*** varexp="", **const char*** select="", **Option_t*** option="")

  Compute the correlation matrix on the attributes given by *varexp* applying the filter contained in *select*. When the parameter *varexp* is empty, the correlation matrix is calculated on all the attributes in the `TDataServer`. The filter *select* is added to the permanent selection of the `TDataServer`. By default, when the *option* is empty, the correlation matrix was calculated on the values (**Pearson matrix**).

  Tip
  The possible values of the argument *option* are:

  rank the correlation was calculated on the ranks (**Spearmann matrix**).

II.5 Visualisation dedicated to uncertainties

ROOT integrates many high level visualisation features, but some of them, devoted to statistics are missing. As they are linked to data, it seemed relevant to develop them in this library. Many of the methods discussed throughout this section are in any case, based on the original ROOT methods that produce plots from a `TTree`-object: the `TTree::Draw` method (and subsequently the `TTree::Scan` when dumping on screen the content of the `TDataServer`).

Summary: `TTree::Draw` and `TTree::Scan` methods

- `TTree::Draw`(**const char*** varexp="", **const char*** select="", **Option_t*** option="")

  Draw the expression *varexp* for specified entries.

  Returns -1 in case of error or number of selected events in case of success.

  The list of the options is located on the THistPainter class on the ROOT website.

- `TTree::Scan`(**const char*** varexp="", **const char*** select="", **Option_t*** option="")

  Scan the expression *varexp* for specified entries.

  Returns -1 in case of error or number of selected events in case of success.

These tree arguments are the same on the `TDataServer::draw` and `TDataServer::scan` methods.

II.5.1 Histogram

The histogram is present in ROOT but it needs to be encapsulated when the user wants to choose an automatic method determining the number of **bins**. By default, the number of bins is given in the variable of the configuration file of ROOT `.rootrc` of the directory `$ROOTSYS/etc`.This information can be overloaded by an user file `.rootrc` in its *home directory ($HOME)*, or in a local file where ROOT is executed. Several methods exist to determine the number of "bins" according to the characteristics of the variable to be visualised.

Again we consider the `TDataServer` built from the `geyser.dat` file, to which we add the attribute *xnorm*, then the histograms will be plotted using the different methods of this new attribute.

```
TCanvas *c2 = new TCanvas();
c2->Divide(2,2);

TDataServer * tdsGeyser = new TDataServer("tdsgeyser", "Databae of the geyser");
tdsGeyser->fileDataRead("geyser.dat");
tdsGeyser->addAttribute("xnorm", "sqrt(x1*x1+x2*x2)");

c2->cd(1); tdsGeyser->draw("xnorm","","nclass=root");
c2->cd(2); tdsGeyser->draw("xnorm","","nclass=sturges");
c2->cd(3); tdsGeyser->draw("xnorm","","nclass=fd");
c2->cd(4); tdsGeyser->draw("xnorm","","nclass=scott");
```

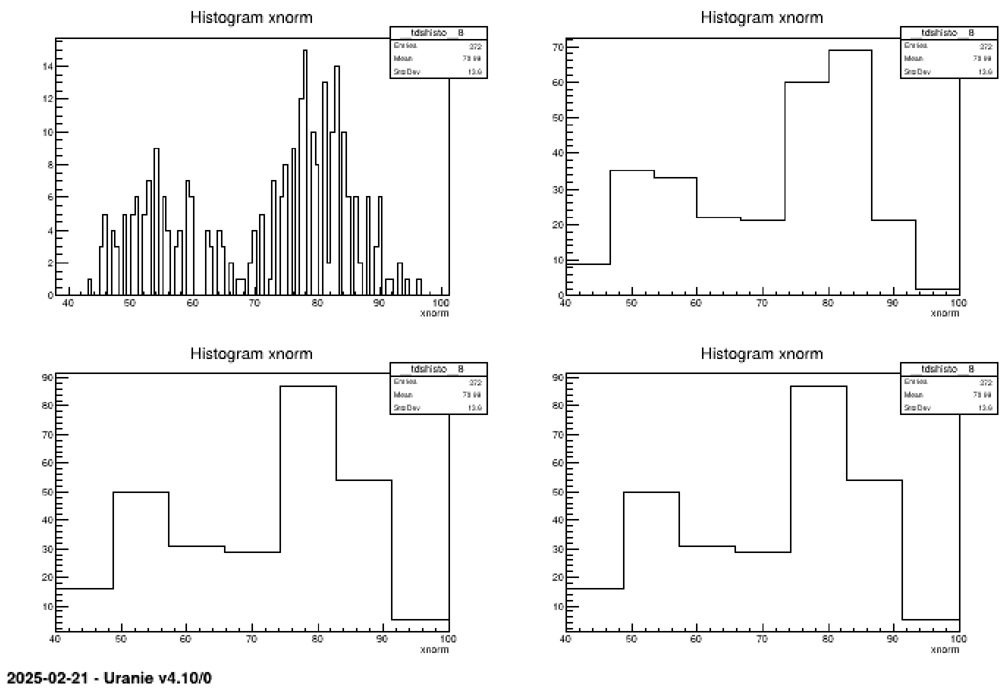


Figure II.42: Different histograms of the same attribute *xnorm* depending on the method for computing bins. The values are respectively 100(root), 8 from sturges, 7 from fd and scoot.

II.5.2 Box-and-whisker("boxplot")

```
tdsGeyser->drawBoxPlot("x2");
```

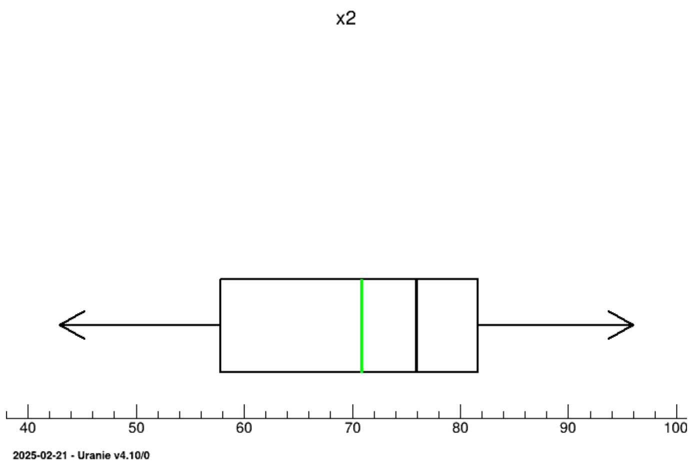


Figure II.43: Boxplot of attribute *x2* of the `TDataServer` `geyser`

II.5.3 CDF, CCDF curves

We can plot the graphs of CDF and/or CCDF.

```
tdsGeyser->drawCDF("x2");
tdsGeyser->drawCDF("x2","","ccdf");
```

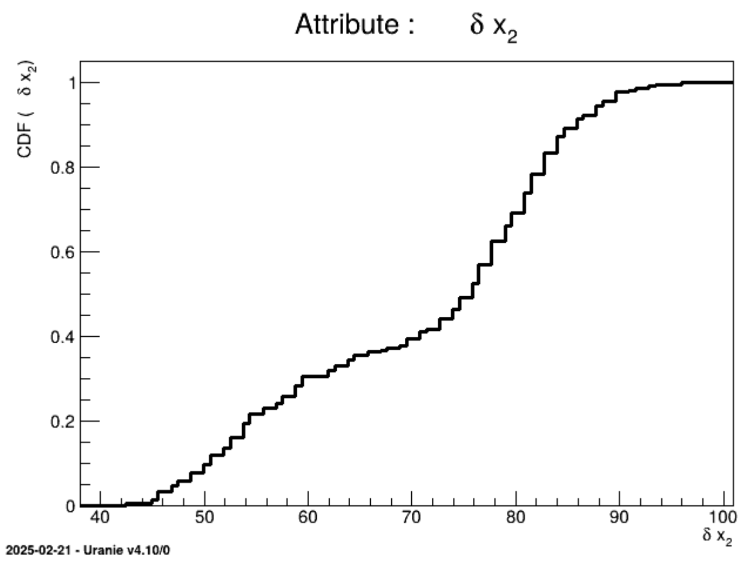


Figure II.44: CDF graph of attribute *x2* of the `TDataServer` `geyser`

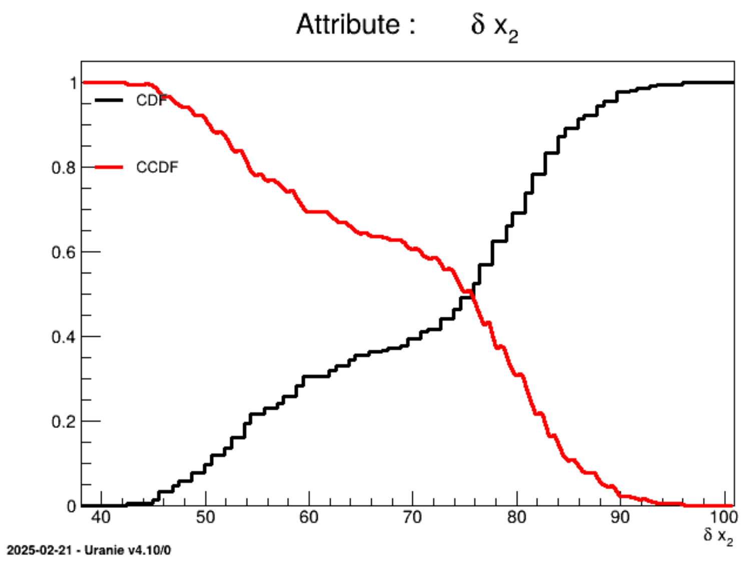


Figure II.45: Graphs CDF+CCDF of the attribute *x2* of the `TDataServer` `geyser`

II.5.4 Graph 2D with contour levels

When a 2D scatterplot is plotted, we do not automatically see the number of points falling in the same cell. To get an assessment of this information, the contour levels of the number of points have to be put in background and then it becomes possible to plot the classical scatterplot.

```
TDataServer * tdsGeyser = new TDataServer("tdsgeyser", "Geyser database");
tdsGeyser->fileDataRead("geyser.dat");
tdsGeyser->drawScatterplot("x2:x1");
```

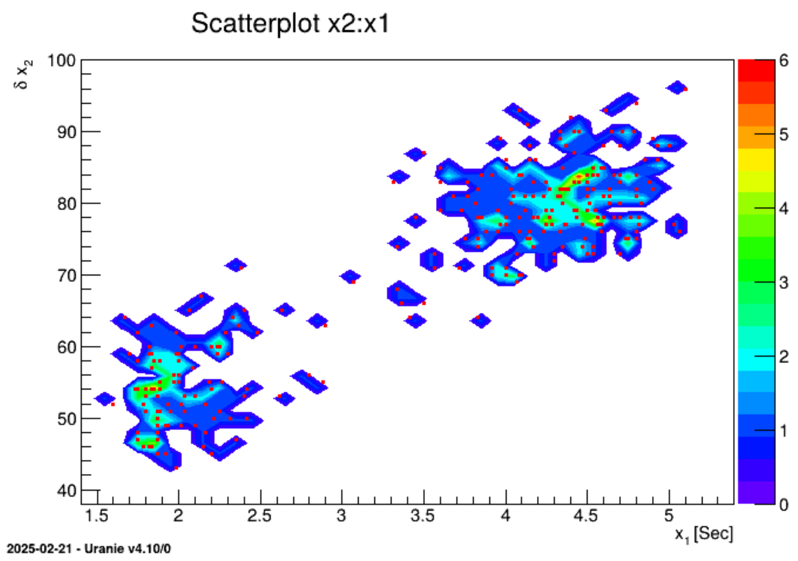


Figure II.46: Scatterplot between attributes x1 and x2 of the `TDataServer` `geyser`.

This figure has to be compared to the classical scatterplot shown, for instance, in Figure II.4.

II.5.5 Graph 2D *"profile"*

It consists in partitioning the "x" axis with a number of *bins* and in plotting, for each segment, the mean value in blue and the standard deviation by a black line. The number of segments N is passed as an option with the formalism **nclass=N**. We can also visualise the scatterplot just below the profile plot, by adding the option "same" (resulting in the red points also shown in Figure II.47).

```
TDataServer * tdsGeyser = new TDataServer("tdsgeyser", "Geyser database");
tdsGeyser->fileDataRead("geyser.dat");
tdsGeyser->drawProfile("x2:x1","","same");
```

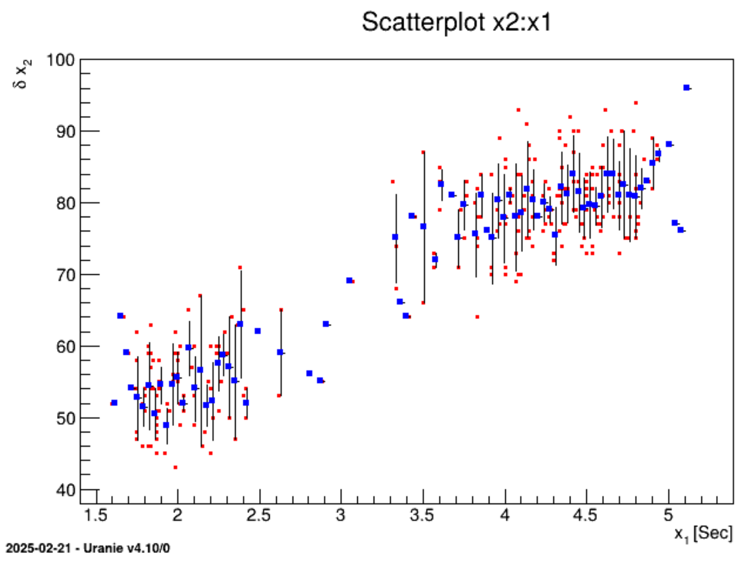


Figure II.47: Scatterplot between attributes x1 and x2 of the `TDataServer` `geyser`.

II.5.6 Graph 2D *"Tufte"*

This 2D graph consists in plotting the scatterplot of the two attributes , and also plotting each of the two histograms of the attributes in X and Y, respectively below or on the left of the scatterplot.

```
TDataServer * tdsGeyser = new TDataServer("tdsgeyser", "Database of the geyser");
tdsGeyser->fileDataRead("geyser.dat");
tdsGeyser->drawTufte("x2:x1");
```

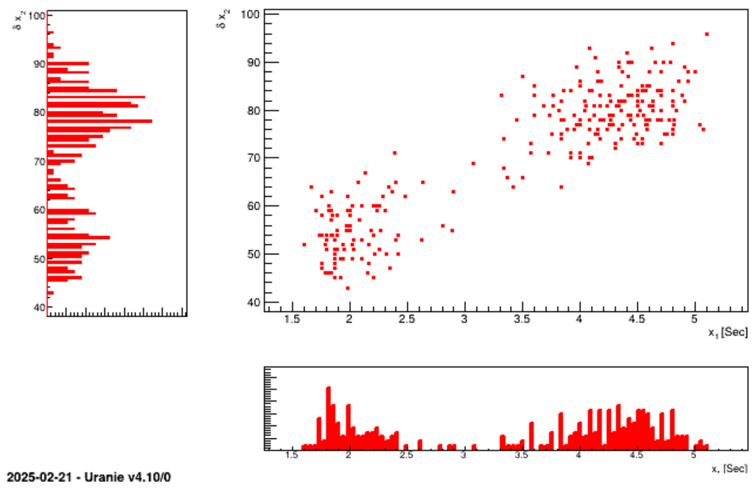


Figure II.48: Graphs of "Tufte" type between the attributes x1 and x2 of the `TDataServer` `geyser`.

II.5.7 Graph 2D *"pairs"*

This 2D graph consists in creating a matrix of graphs, where for the (i,j) cells with i different from j, the graph contains the scatterplot of the attribute j versus the attribute i, and for cell (i,i), the graph contains the histogram of the attribute i.

```
TDataServer * tdsGeyser = new TDataServer("tdsgeyser", "Database of the geyser");
tdsGeyser->fileDataRead("flowrateUniformDesign.dat");
tdsGeyser->drawPairs();
```

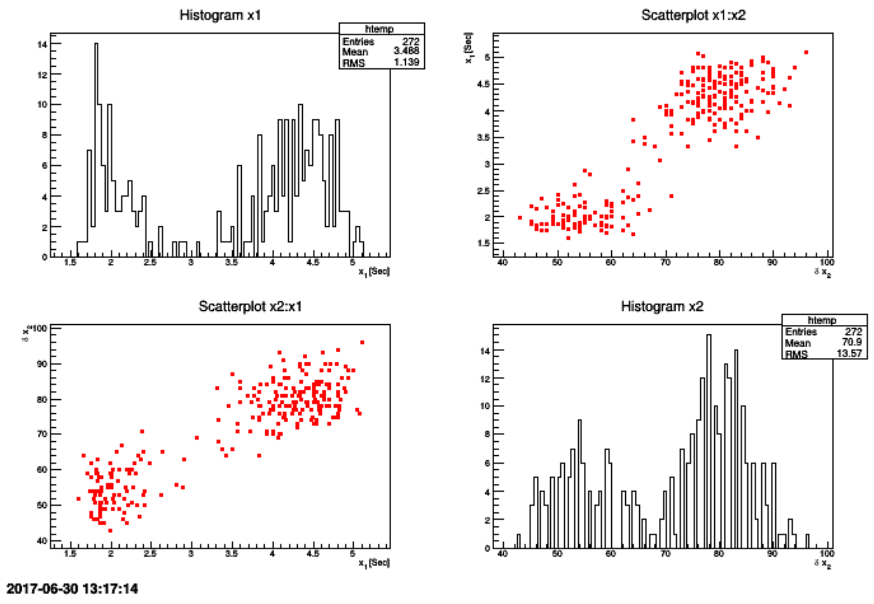


Figure II.49: Graphs of "Tufte" type between the attributes x1 and x2 of the `TDataServer` `geyser`.

Summary: 2D Graph

- `drawScatterplot` (**const char*** varexp, **const char*** selection="", **Option_t*** option="")

  Draw 2D graph with the number of points of attributes located in *varexp* on the background .

- `drawProfile` (**const char*** varexp, **const char*** selection="", **Option_t*** option="")

  Draw 2D graphic of the mean and standard deviation of each segment of "X" axis for the attributes contained in *varexp*.

  Tip
  The possible values of options *option* are:

  nclass=[0-9]* Specifies the number of segments in "X" axis .

  same Displays the 2D scatterplot below "profile".

- `drawTufte` (**const char*** varexp, **const char*** selection="", **Option_t*** option="")

  Draw 2D graph of the attributes located in *varexp*.

  Tip
  The possible values of options *option* are:

  optstat Prints the window containing statistics in each histogram

  scatter Prints the scatterplot the same way as the command drawScatterplot.

- `drawPairs` (**const char*** varexp, **const char*** selection="", **Option_t*** option="")

  Create the matrix of graph with the attributes located in *varexp*.

It should be noticed that these functions have the same signature as the `draw` method of a **TTree** of ROOT.

Warning
For all these methods, the character *varexp* must contain either two attributes or a single character ":"

II.5.8 Graph *"CobWeb"*

For multidimensional problem, the `drawPairs` method is limited to spot correlation because of the way the output looks. For instance in a problem with 8 uniformly-distributed inputs and one output, one can get a graphic as the one shown in Figure II.50 (obtained with the code below). No special trend can be seen here.

```
TDataServer * tdsCobweb = new TDataServer("tdscobweb", "Database of the cobweb");
tdsCobweb->fileDataRead("cobwebdata.dat"); // read data file
tdsCobweb->drawPairs(); // do the drawPairs graph
```

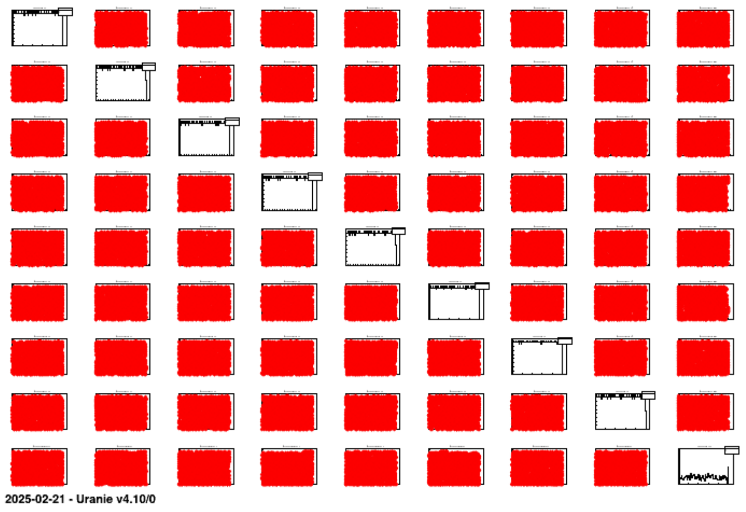


Figure II.50: Graphs of "drawPairs" type between the 8 uniformly-distributed inputs and the output of a given problem.

The "CobWeb" multidimensional graph, on the other hand, consists in plotting every dimension on a vertical axis and connecting all the points for a single event with a straight line. It is particularly useful to spot correlation in high-dimension problems as it is simple to highlight a certain region (by changing the colour for instance) to see if the rest of the variable are randomly distributed or not.

```
// Draw the cobweb plot
tdsCobweb->drawCobWeb("x0:x1:x2:x3:x4:x5:x6:x7:out"); // Draw the cobweb

// Get the parallel coordination part to perform modification
TParallelCoord* para = (TParallelCoord*)gPad->GetListOfPrimitives()->FindObject(" ↩
    __tdspara__1");
// Get the output axis
TParallelCoordVar* axis = (TParallelCoordVar*)para->GetVarList()->FindObject("out");

// Create a range for 0.97 < out < 1.0 and display it in blue
TParallelCoordRange* Range  = new TParallelCoordRange(axis,0.97,1.0);
axis->AddRange(Range);
para->AddSelection("blue");
Range->Draw();
```

This code for instance specifies a region-of-interest in the output, when the value are greater than 0.97. A correlation is highlighted between these values and a region, for which the fourth input variable has high value while the sixth input variable values lie between 0.2 and 0.4. The result of this code is shown in Figure II.51.

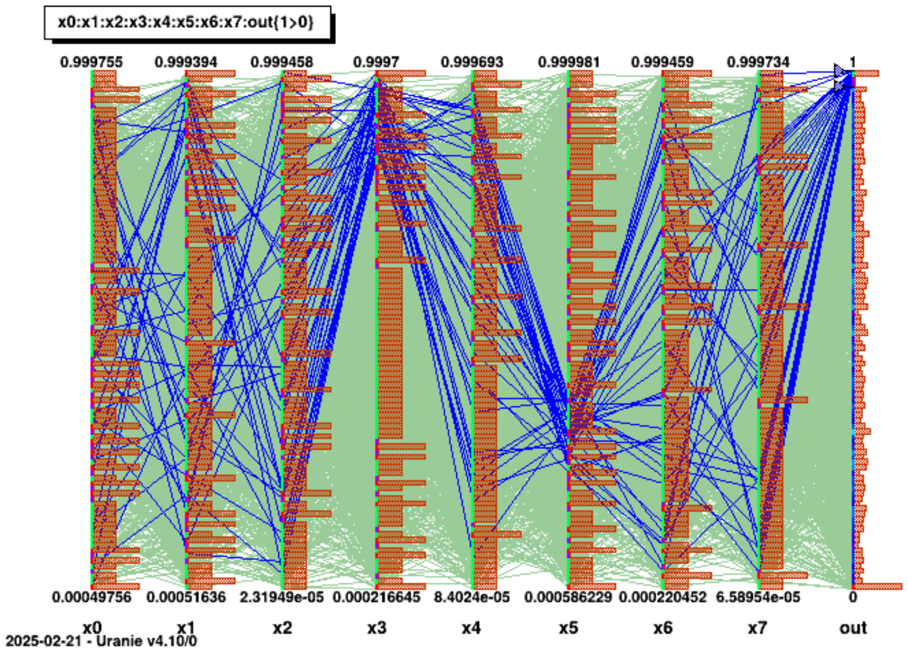


Figure II.51: Graphs of "CobWeb" type between the 8 uniformly-distributed inputs and the output of a given problem.

II.5.9 *QQ* plot

Warning
This function requires the "mathmore" feature to have be installed along with your ROOT version. If not found, this function cannot be used and will return nothing but an equivalent to this message.

Once dealing with an unknown set of points, it is possible to compare it to known statistical law (among the following already implemented list: normal, uniform, weibull, gumbelmax, exponential, beta, gamma and lognormal). For instance, if one wants to compare "x2" variable from the geyser dataset to a normal law, one can have a look at the quantile distribution from the sample and compare it to the expected behaviour by following the steps below.

```
TDataServer * tdsQQ = new TDataServer("tdsQQ", "Database of the QQ");
tdsQQ->fileDataRead("geyser.dat"); // read data file
tdsQQ->computeStatistic(); // to estimate mu and sigma
tdsQQ->drawQQPlot("x2",Form("normal(%g,%g)",tdsQQ->getAttribute("x2")->getMean(),tdsQQ-> ↩
    getAttribute("x2")->getStd()),400);
```

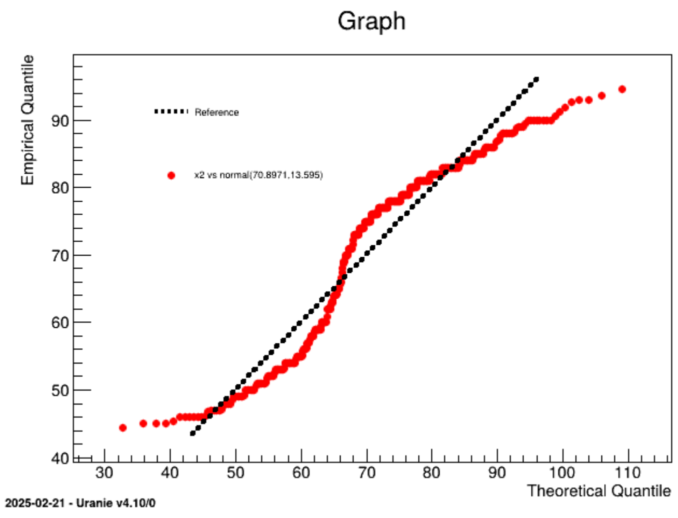


Figure II.52: Plot resulting from the "drawQQPlot" method, comparing "x2" to a normal distribution.

It is obvious here, that the "x2" law, clearly doesn't seem to follow a normal law (which was pretty obvious by looking at Figure II.48 for instance). An heavy use of this method is provided in Section XIV.2.15.

II.5.10 *PP* plot

Warning
This function requires the "mathmore" feature to have be installed along with your ROOT version. If not found, this function cannot be used and will return nothing but an equivalent to this message.

Once dealing with an unknown set of points, it is possible to compare it to known statistical law (among the following already implemented list: normal, uniform, weibull, gumbelmax, exponential, beta, gamma and lognormal). For instance, if one wants to compare "x2" variable from the geyser dataset to a normal law, one can have a look at the probabiliity distribution from the sample and compare it to the expected behaviour by following the steps below.

```
TDataServer * tdsPP = new TDataServer("tdsPP", "Database of the PP");
tdsPP->fileDataRead("geyser.dat"); // read data file
tdsPP->computeStatistic(); // to estimate mu and sigma
tdsPP->drawPPPlot("x2",Form("normal(%g,%g)",tdsPP->getAttribute("x2")->getMean(),tdsPP-> ↩
    getAttribute("x2")->getStd()),400);
```

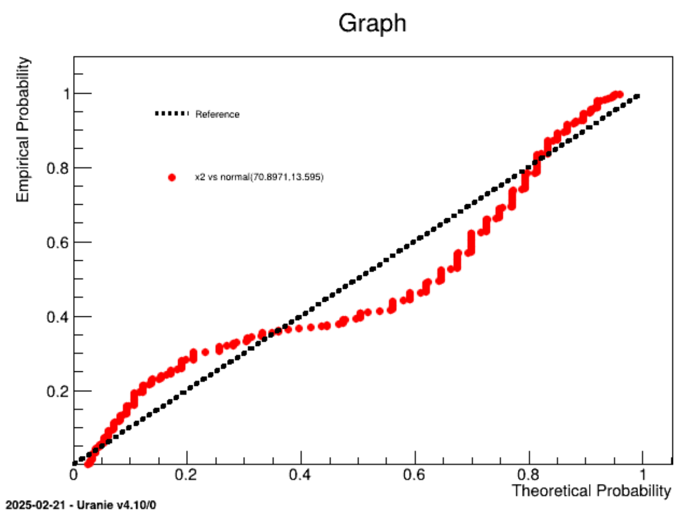


Figure II.53: Plot resulting from the "drawPPPlot" method, comparing "x2" to a normal distribution.

It is obvious here, that the "x2" law, clearly doesn't seem to follow a normal law (which was pretty obvious by looking at Figure II.48 for instance). An heavy use of this method is provided in Section XIV.2.16.

II.6 Combining these aspects: performing PCA

This part is introducing an example of analysis that combines all the aspects discussed up to now: handling data, perform a statistical treatment and visualise the results. This analysis is called PCA for *Principal Component Analysis* and is often used to

- gather event in a sample that seem to have a common behaviour;
- reduce the dimension of the problem under study.

There is a very large number of articles, even books, discussing the theoretical aspects of principal component analysis (for instance one can have a look at [18]) a small theoretical introduction can in any case be found in [30].

II.6.1 PCA usage within Uranie

Let's use a relatively simple example to illustrate the principle and the way we can achieve a reduction of dimension while keeping the inertia as large as possible. One can have a look at a sample of marks from different pupils, in various kinds of subject, all gathered in the `Notes.dat` whose content is shown below.

```
#TITLE: Marks of my pupils
#COLUMN_NAMES: Pupil | Maths | Physics | French | Latin | Music
#COLUMN_TYPES: S|D|D|D|D|D

Jean 6 6 5 5.5 8
Aline 8 8 8 8 9
Annie 6 7 11 9.5 11
Monique 14.5 14.5 15.5 15 8
Didier 14 14 12 12 10
Andre 11 10 5.5 7 13
Pierre 5.5 7 14 11.5 10
Brigitte 13 12.5 8.5 9.5 12
Evelyne 9 9.5 12.5 12 18
```

One can have a look at some of these variables against one another to have a sense of what's about to be done. In Figure II.54, the maths marks of the pupils are displayed against latin on the left and physics on the right. Whereas no specific trend is shown on the left part, an obvious correlation can be seen from the right figure meaning one can try extrapolate the maths marks from the value of the physics one.

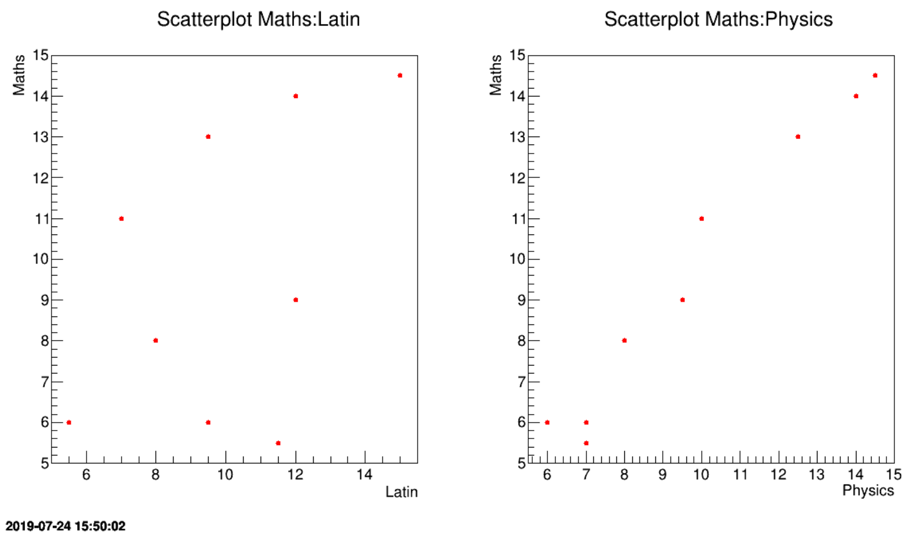


Figure II.54: Representation of some variables of the Notes sample.

II.6.1.1 Perform the PCA

The way to perform this analysis is rather simple through Uranie: simply provide the dataserver that contains data to an `TPCA` object, only precising the name of the variables to be investigated. This is exactly what's done below:

```
 // Read the database
TDataServer * tdsPCA = new TDataServer("tdsPCA", "my TDS");
tdsPCA->fileDataRead("Notes.dat");

// Create the PCA object precising the variables of interest
TPCA * tpca = new TPCA(tdsPCA, "Maths:Physics:French:Latin:Music");
tpca->compute();
```

Once done, the process described in the [30] is finished and then interpretation starts.

II.6.1.2 Interpretation of PCA results

The first result that one should consider is the non-zero eigenvalues (so the variance of the corresponding principal components). These values are stored in a specific ntuple, that can be retrieve by calling the method `getResultNtupleD(`
This ntuple contains three kind of information: the eigenvalues itself (*"eigen"*), these values as contributions in percent of the sum of eigenvalues (*"eigen_pct"*) and the sum of these contributions (*"sum_eigen_pct"*). All these values can be accessed through the usual `Scan` method (see Section XIV.2.17 to see the results).

The following lines are showing how to represent the eigenvalues as plots and the results are gathered in Figure II.55. From this figure, one can easily conclude that only the three first principal components are useful (as it reached an inertia of a bit more than 99% of the original one). This is the solution chosen for the rest of the graphical representations below and this is the first nice results from this step: it is possible to reduce the dimension of our problem from the 5 different subjects to the three principal component that needs to be explicated.

```
// Draw the eigen values in different normalisation
TCanvas *c = new TCanvas("cEigenValues", "Eigen Values Plot",1100,500);
TPad *apad3 = new TPad("apad3","apad3",0, 0.03, 1, 1); apad3->Draw(); apad3->cd();
apad3->Divide(3,1);
TNtupleD *ntd = tpca->getResultNtupleD();
apad3->cd(1); ntd->Draw("eigen:i","","lp");
apad3->cd(2); ntd->Draw("eigen_pct:i","","lp"); gPad->SetGrid();
apad3->cd(3); ntd->Draw("sum_eigen_pct:i","","lp"); gPad->SetGrid();
c->SaveAs("PCA_notes_musique_Eigen.png");
```

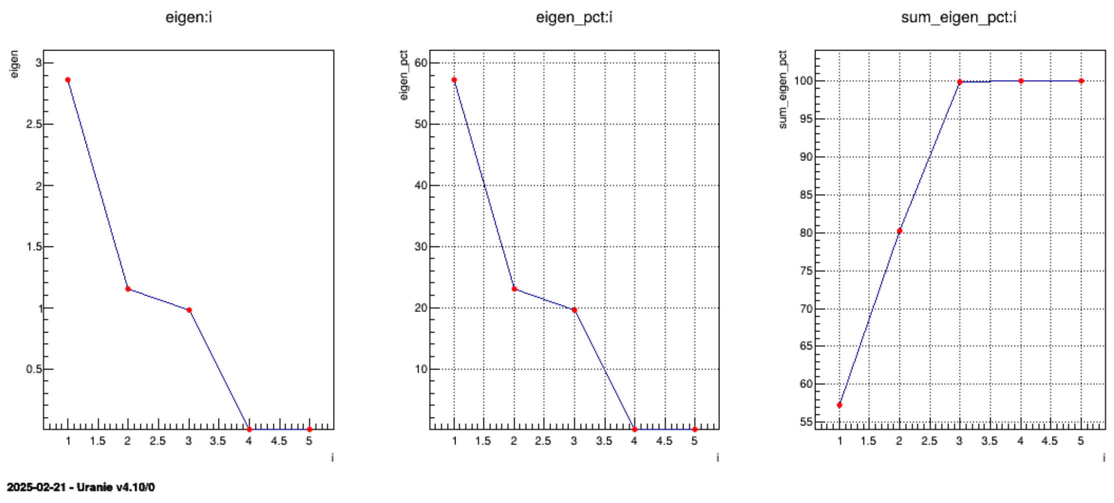


Figure II.55: Representation of the eigenvalues (left) their overall contributions in percent (middle) and the sum of the contributions (right) from the PCA analysis.

As stated previously, one can have a look at the variables in their usual representation (that can be seen in Figure II.56) that is called the correlation circle. This plot is obtained by calling the `drawLoading` method that takes two arguments: the first one being the number of the PC used as x-axis while the second one is the number of the PC used as y-axis. The code to get Figure II.56 is shown below:

```
// Draw all variable weight in PC definition
TCanvas *cLoading = new TCanvas("cLoading", "Loading Plot",800,800);
TPad *apad2 = new TPad("apad2","apad2",0, 0.03, 1, 1); apad2->Draw(); apad2->cd();
apad2->Divide(2,2);
apad2->cd(1); tpca->drawLoading(1,2);
apad2->cd(3); tpca->drawLoading(1,3);
apad2->cd(4); tpca->drawLoading(2,3);
```

Figure II.56 represents the correlation between the original variable and the principal component under study. Two kinds of principal component are resulting from PCA:

- the size one where all variables are on the same size of the principal component. In our case, the first PC in Figure II.56 represents the variability of marks.
- the shape one where variables are balanced from positive to negative value of the principal component. In our case, the second PC in Figure II.56 represents the difference observed between scientific variables, such as maths and physics, from literary ones, such as french and latin.

Finally to complete the picture, the third PC is a shape one that represents the fact that music seems to have a very specific behaviour, different from all the other studies.

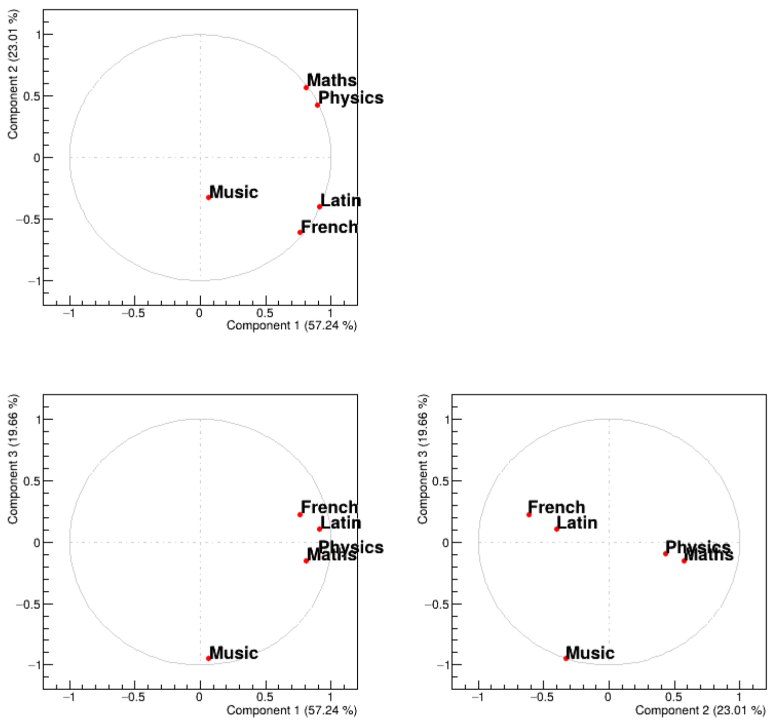


Figure II.56: Representation of correlation between the original variables and the PC under study.

Finally, one can also have a look at the points distribution in the new PC space (that can be seen in Figure II.57). This plot is obtained by calling the method `drawPCA` that takes two compulsory arguments: the first one being the number of the PC used as x-axis while the second one is the number of the PC used as y-axis. An extra argument can be provided that is used to specify a branch in the database that contains the name of the points (the name of the pupils) which allows to get a nice final plot. The code to get Figure II.57 is shown below:

```
// Draw all point in PCA planes
TCanvas *cPCA = new TCanvas("cpca", "PCA",800,800);
TPad *apad1 = new TPad("apad1","apad1",0, 0.03, 1, 1); apad1->Draw(); apad1->cd();
apad1->Divide(2,2);
apad1->cd(1); tpca->drawPCA(1,2,"Pupil");
apad1->cd(3); tpca->drawPCA(1,3,"Pupil");
apad1->cd(4); tpca->drawPCA(2,3,"Pupil");
```

Figure II.57 shows where to find our pupils with respect to our new basis and the interpretation of the correlation plot of the variable holds as well:

- looking at PC1, it seems to be defined by two extremes, Jean and Monique, the former being bad at every subjects while the latter is good at (almost) all subjects.
- looking at PC2, it also seems to be defined by two extremes, Pierre and Andre, the former having way better marks at literary subjects than at scientific ones while its the other way around for the latter.

- looking at PC3, it seems to oppose Evelyn to all the other pupils, and this correspond to fact that every pupils is bad at music, but Evelyn.

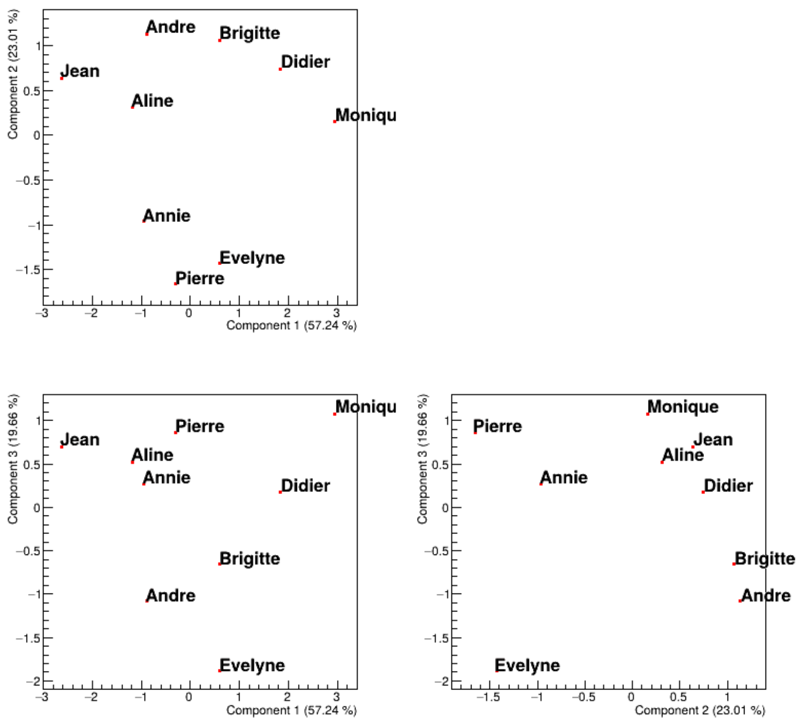


Figure II.57: Representation of the data points in the PC-defined plane.

All these interpretations remain graphical and give an easy way to summarise the impact of the PCA analysis. Disregarding the sources considered (the variables or the points), one can have quantitative interpretation coefficients that detail more the "feeling" described above:

- the quality of the representation: it shows how well the source (either the subject or the pupil in our case) is represented by a given PC. Summing these values over the PC should lead to 1 for every sources.
- the contribution to axis: it shows how much the source (either the subject or the pupil in our case) contribute to the definition of the given PC. Summing these values over the source (for the given PC) should lead to 1 for every PC.

Example of how to get these numbers can be found in the use-case macros, see Section XIV.2.17.

Chapter III

The Sampler module

III.1 Introduction

The Sampler module is used to produce design-of-experiments knowing the expected behaviour of the input variables for the problem under consideration. The framework of our approach can be illustrated in the following schematic view:

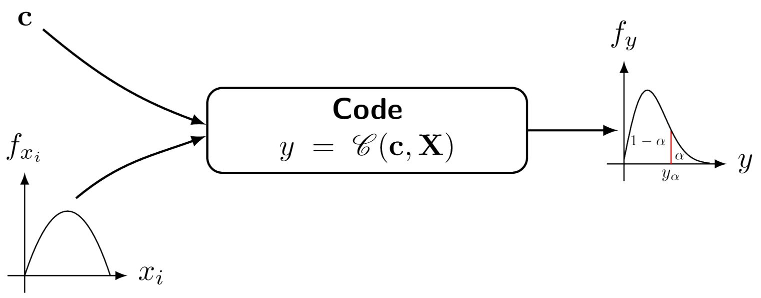


Figure III.1: Schematic view of the input/output relation through a code

- We will denote as $\mathcal{C}$ the studied computational code which, generally, has two types of inputs:
  - The **constant** parameters which are gathered in the vector $\mathbf{c} \in \mathbb{R}^{n_C}$. They represent constants.
  - The **uncertain** parameters which are gathered in the vector $\mathbf{X} \in \mathbb{R}^{n_X}$
    It shall be noticed that these parameters are supposed to be **uncertain** either because of a lack of knowledge on their actual value or because of their intrinsic random nature.
- The result of the code $\mathcal{C}$ for a given set of parameters $(\mathbf{c},\mathbf{X})$ gives the vector $y \in \mathbb{R}^{n_Y} = \mathcal{C}(\mathbf{c},\mathbf{X})$ which contains all the output variables of the analysis.

Different methods exist to obtain a design-of-experiments from uncertain parameters which can be classified into two categories:

1. **stochastic** methods (see Section III.2). These methods consist in using a random number generator to produce new samples. This is also called Monte-Carlo.
2. **deterministic** methods (see Section III.4). Two distinct calls with the same parameters will always give the same point in a design-of-experiments. Some of these methods (those discussed below) are sequences which are sometimes called quasi-Monte Carlo (qMC).

III.2 The Stochastic methods

III.2.1 Introduction

The stochastic classes all inherit from the `TSampler` class through the `TSamplerStochastic` one. In these classes the knowledge (or mis-knowledge) of the model is encoded in the choice of probability law used to describe the inputs x_i, for $i \in [0, n_X]$. These laws are usually defined by:

- a range that describes the possible values of x_i
- the nature of the law, which has to be taken in the list of `TStochasticAttribute` already presented in Section II.2.5

A choice has frequently to be made between two implemented methods of drawing:

SRS (Simple Random Sampling): This method consists in independently generating the samples for each parameter following its own probability density function. The obtained parameter variance is rather high, meaning that the precision of the estimation is poor leading to a need for many repetitions in order to reach a satisfactory precision. An example of this sampling when having two independent random variables (uniform and normal one) is shown in Figure III.3-left.

LHS (Latin Hypercube Sampling): this method [4] consists in partitioning the interval of each parameter so as to obtain segments of equal probabilities, and afterwards in selecting, for each segment, a value representing this segment. An example of this sampling when having two independent random variables (uniform and normal one) is shown in Figure III.3-right.

Both methods consist in generating a N_{calc}-sample represented by a matrix U called **matrix of the design-of-experiments**. The number of columns of the matrix U correspond to the number of uncertain parameters n_X, and the number of lines is equal to the size of the sample N_{calc}.

The first method is fine when the computation time of a simulation is "satisfactory". As a matter of fact, it has the advantage of being easy to implement and to explain; and it produces estimators with good properties not only for the mean value but also for the variance. Naturally, it is necessary to be careful in the sense to be given to the term "satisfactory". If the objective is to obtain quantiles for extreme probability values α (*i.e.* α = 0.99999 for instance), even for a very low computation time, the size of the sample would be too large for this method to be used. When a computation time becomes important, the LHS sampling method is preferable to get robust results even with small-size samples (*i.e.* N_{calc} = 50 to 200) [5]. On the other hand, it is rather trivial to double the size of an existing SRS sampling, as no extra caution has to be taken apart from the random seed.

In Figure III.2, we present two samples of size N_{calc} = 8 coming from these two sampling methods for two random variables U_1 according to a gaussian law, and U_2 a uniform law. To make the comparison easier, we have represented on both figures the partition grid of equiprobable segments of the LHS method, keeping in mind that it is not used by the SRS method. These figures clearly show that for LHS method each variable is represented on the whole domain of variation, which is not the case for the SRS method. This latter gives samples that are concentrated around the mean vector; the extremes of distribution being, by definition, rare.

Concerning the LHS method (right figure), once a point has been chosen in a segment of the first variable U_1, no other point of this segment will be picked up later, which is hinted by the vertical red bar. It is the same thing for all other variables, and this process is repeated until the N_{calc} points are obtained. This elementary principle will ensure that the domain of variation of each variable is totally covered in a homogeneous way. On the other hand, it is absolutely not possible to remove or add points to a LHS sampling without having to regenerate it completely. A more realistic picture is draw in Figure III.3 with the same laws, both for SRS on the left and LHS on the right. The "tufte" representation (presented in Section II.5.6) clearly shows the difference between both methods when considering one-dimensional distribution.

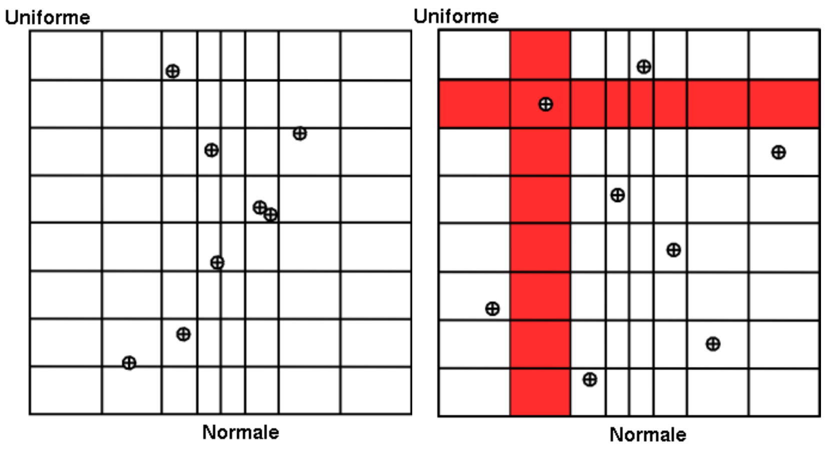


Figure III.2: Comparison of the two sampling methods SRS (left) and LHS (right) with samples of size 8.

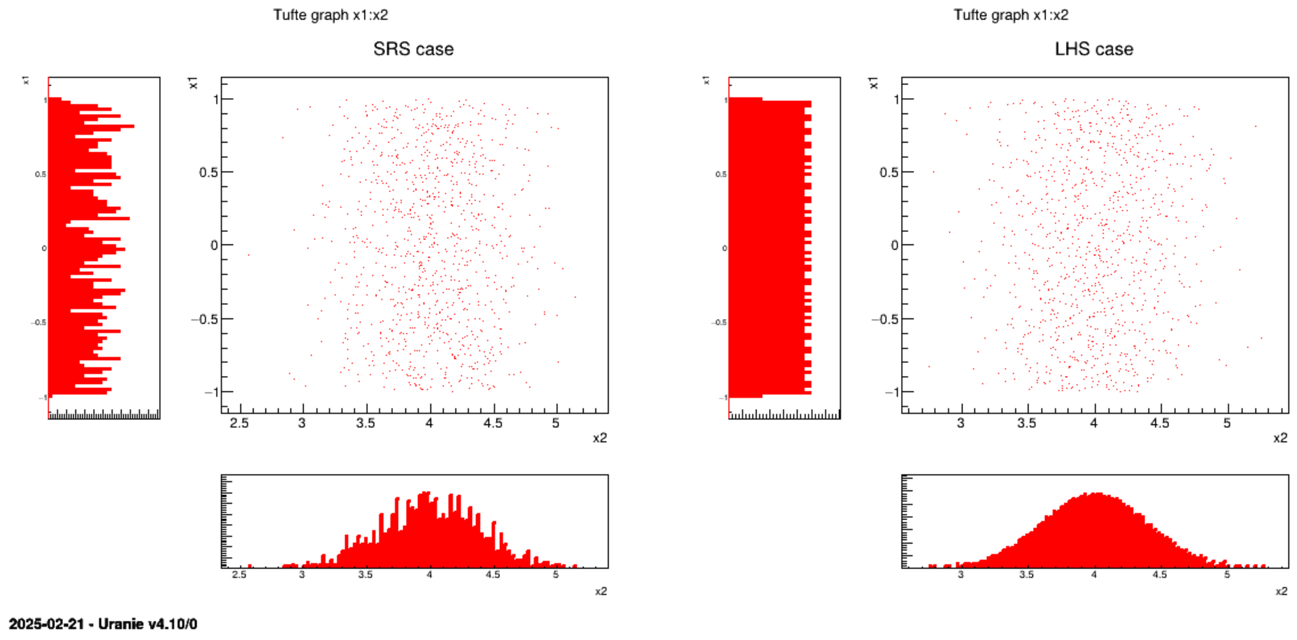


Figure III.3: Comparison of deterministic design-of-experiments obtained using either SRS (left) or LHS (right) algorithm, when having two independent random variables (uniform and normal one)

There are two different sub-categories of LHS design-of-experiments discussed here and whose goal might slightly differs from the main LHS design discussed above:

- the maximin LHS: this category is the result of an optimisation whose purpose is to maximise the minimal distance between any sets of two locations. This is discussed later-on in Section III.2.2.
- the constrained LHS: this category is defined by the fact that someone wants to have a design-of-experiments fulfilling

all properties of a Latin Hypercube Design but adding one or more constraints on the input space definition (generally inducing correlation between varibles). This is also further discussed in Section III.2.2 and in Section III.2.4.

Once the nature of the law is chosen, along with a variation range, for all inputs x_i, the correlation between these variables has to be taken into account. This is further discussed in Section III.3

III.2.2 The main sampler classes

The sampler classes are built with the same skeleton. They consist in a three steps procedure: `init`, `generateSample` and `terminate`. There are five different types of Stochastic sampler that can be used:

- **`TSampling`:** general sampler where the sampling is done with the Iman and Conover methods ([6]) that uses a double Cholesky decomposition in order to respect the requested correlation matrix. This methods has the drawback of asking a number of samples at least twice as large as the number of attribute.
- **`TBasicSampling`:** very simple implementation of the random sampling that can, as well, produce stratified sampling. As it is fairly simple, there is no lower limit in the number of random samples that can be generated and generation is much faster when dealing with a large number of attributes than with the `TSampling`. On the other hand, even though a correlation can be imposed between the variables, it will be done in a simple way that cannot by construction respect the stratified aspect (if requested, see [30] explanations).
- **`TMaxiMinLHS`:** class recently introduced to produced maximin LHS, whose purpose is to get the best coverage of the input space (for a deeper discussion on the motivation, see [30]). It can be used with a provided LHS drawing or it can start from scratch (using the `TSampling` class to get the starting point). The result of this procedure can be seen in Figure XIV.13 which was procuded with the macro introduced in Section XIV.3.11;

Tip
The optimisation to get a maximin LHS is done on the mindist criterion: let $D = [\mathbf{x}_1, \cdots, \mathbf{x}_N] \subset [0,1]^d$ be a design-of-experiments with N points. The mindist criterion is written as: $\min_{i,j} ||\mathbf{x}_i - \mathbf{x}_j||_2$ where $||.||_2$ is the euclidian norm.
This information can be computed on any grid, by calling the static function

```
double mindist = TMaxiMinLHS::getMinDist(TMatrixD LHSgrid);
```

- **`TConstrLHS`:** class recently introduced to produced constrained LHS, whose purpose is to get nicely covered marginal laws while respecting one or more constraints imposed by the user. It can be used with a provided LHS drawing or it can start from scratch (using the `TSampling` class to get the starting point) and it is discussed a bit further in Section III.2.4. The result of this procedure can be seen in Figure XIV.14 which was procuded with the macro introduced in Section XIV.3.12;
- **`TGaussianSampling`:** very basic sampler specially done for cases composed with only normal distributions. It relies on the drawing of a sample $A(n_S, n_X)$ n_S being the number of sample and n_X the number of inputs. The sample is transformed to account for correlation (see [30]) and shifted by the requested mean (for every variables).
- **`TImportanceSampling`:** use the same method as `TSampling` but should be used when one `TAttribute` has a son. In this case the range for the considered attribute is the sub-range used to define the son and there is specific reweighting procedure to take into account the fact that the authorised range is reduced. LHS is therefore used by default.

III.2.3 Simple example

The following piece shows how to generate a sample using the `TSampling` class.

```
TUniformDistribution *xunif = new TUniformDistribution("x1", 3., 4.);      ❶[1]
TNormalDistribution  *xnorm = new TNormalDistribution("x2", 0.5, 1.5);     ❷[2]

TDataServer * tds = new TDataServer("tdsSampling", "Demonstration Sampling");
tds->addAttribute(xunif);
tds->addAttribute(xnorm);

// Generate the sampling from the TDataServer object
TSampling *sampling = new TSampling(tds, "lhs", 1000); ❸[3]
sampling->generateSample(); ❹[4]
tds->drawTufte("x2:x1"); ❺[5]
```

Generation of a design-of-experiments with stochastic attributes

❶ Generating a uniform random variable in [3., 4.]

❷ Generating a gaussian random variable with a mean of 0.5 and a standard deviation of 1.5

❸ Construct the sampler object requesting a sample of 1000 events, using the Lhs algorithm.

❹ Method to generate the sample.

❺ Draw the plot as the right side of Figure III.3

III.2.4 TConstrLHS example

This section will discuss the way to produce a constrained LHS design-of-experiments from scratch, focusing on the main problematic part: defining one or more constraints and on which variable to apply them. The logic behind the heuristic is supposed to be known, so if it's not the case, please have a look at the dedicated section in [30]. This section will mostly rely on the way to define the constraints and the variables on which these should be applied on which are specified by the method `addConstraint`. This methods takes 4 arguments:

1. a pointer to the function that will compute the constraints values;
2. the number of constraint defined in the function discussed above;
3. the number of parameters that are provided to the function discussed above;
4. the values of the parameters that are provided to the function discussed above;

The main object is indeed the constraint function and the way it is defined is discussed here. It is a C++ function (for convenience as the platform is C++-coded) but this should not be an issue even for python users. The followings lines are showing the example of the constraint function used to produce the plot in Section XIV.3.12.

```
void Linear(double *p, double *y)
{
    double p1=p[0], p2=p[1];
    double p3=p[2], p4=p[3];

    // Linear constaint
    y[0] = (( (p1 + p2>=2.5) || (p1-p2<=0) ) ? 0 : 1);
    y[1] = ((p3 - p4<0) ? 0 : 1);

}
```

Here are few elements to discuss and explain this function:

- its prototype is the usual C++-ROOT one with a pointer to the input parameter `p` and a pointer to the output (here the constraint results) `y`;
- the first lines are defining the parameters, meaning the couples (x_{row}, x_{col}) for all the constraints. By convention, the first element of every line ($p1$ and $p3$) are of the row type (they will not change in the design-of-experiments through this constraint, see [30] for clarification) while the second parameters ($p2$ and $p4$) are of the column type, meaning their order in the design-of-experiments will change through permutations through this constraint.
- the rest of the lines are showing the way to compute the constraints and to interpret them thanks to the trilinear operator (even though the classical `if`, `else` would perfectly do the trick as well. Let's focus first on the second constraint:

  ```
  y[1] = ((p3 - p4<0) ? 0 : 1);
  ```

  if `p3` is lower than `p4` (`p3-p4<0`) then the function will put 0 in `y[1]` (stating that this configuration is not fulfilling the constraint), and it will put 1 otherwise (stating that the constraint is fulfilled). The other line is defining another constraint which is composed of two tests on the same couple of variables:

  ```
  y[0] = (( (p1 + p2>=2.5) || (p1-p2<=0) ) ? 0 : 1);
  ```

  Here, two constraints are combined in once, as they affect the same couple of variables, the configuration will be rejected either if `p1+p2` is greater than 2.5 or if `p1` is lower than `p2`.

Once done, this function needs to be plugged into our code in order to state what variables are `p1`, `p2`, `p3` and `p4` so that the rest of the procedure discussed in [30] can be run. This is done in the `addConstraint` method, thanks to the third and fourth paramaters which are taken from a single object: a `vector<int>` in C++ and a `numpy.array` in python. It defines a list of indices (integers) that corresponds to the number of the input attributes as it is has been added into the `TDataServer` object. For instance in our case, the list of input attributes is `"x0:x1:x2"` while the constraints are coupling (x_1, x_0) and (x_2, x_1) respectively. Once translated in term of indices, the constraints are coupling respectively $(1,0)$ and $(2,1)$, so the list of parameter should reflect this which is shown below:

```
vector<int> inputs = {1,0,2,1};
constrlhs->addConstraint(Linear, 2, inputs.size(), &inputs[0]);
```

Warning The consistency between the function and the list of parameters is up to you and you should keep a carefull watch over it. It is true that an inequality can be written in two ways, as

$$x_1 - x_0 < 0 \;\Leftrightarrow\; x_0 - x_1 > 0$$

but when the constraint is defined as

```
y[0] = ((p[0] - p[1]<0) ? 0 : 1);
```

the results will be drastically different if the list of parameters is $(1,0)$ (which should fulfill the constraint shown above) or $(0,1)$ which would results in the exact opposite behaviour (and might make the heuristic crash if the constraint cannot be fulfilled).

III.3 Description of a correlation

We will describe in this section how to precise in **Uranie** the necessary information to take into account the correlations between variables. There are two ways to specify this: setting the correlation coefficient by hand or couple the two considered attributes with a copula. Both methods are described below.

III.3.1 Imposing the correlation coefficients

It is possible to set the correlation coefficient by hand, using the `setUserCorrelation` method of both `TSampling` and `TBasicSampling`.

```
TUniformDistribution *xunif = new TUniformDistribution("x1", 3., 4.);      ❶[1]
TNormalDistribution  *xnorm = new TNormalDistribution("x2", 0.5, 1.5);     ❷[2]

TDataServer * tds = new TDataServer("tdsSampling", "Demonstration Sampling");
tds->addAttribute(xunif);
tds->addAttribute(xnorm);

// Generate the sampling from the TDataServer object
TSampling *sampling = new TSampling(tds, "lhs", 1000);
sampling->setUserCorrelation( xunif, xnorm, 0.90);     ❸[3]
sampling->generateSample();    ❹[4]
tds->Draw("x2:x1","","");
```

Specification of a correlation

❶ Generating a uniform random variable in [3., 4.]

❷ Generating a gaussian random variable with 0.5 as mean value 1.5 as standard deviation.

❸ Specification of correlation of 0.90 between the two attributes `xunif` and `xnorm`.

❹ Generate the sample by calling the `generateSample` method in `TSampling` (respectively the `generateCorrS` method in `TBasicSampling`).

The coefficient provided is changing the correlation between the two considered attributes (for a more general discussion on this, see [30]). As an example, Figure III.4 and Figure III.5 show the design-of-experiments created with the `TSampling` class, using a normal and uniform distribution, using three different correlation coefficient values: 0, 0.45 and 0.9. The first one is showing the obtained 2D plan and their corresponding projections, while the second one is showing the same information but displaying their rank instead of their values (this is further discussed in [30]).

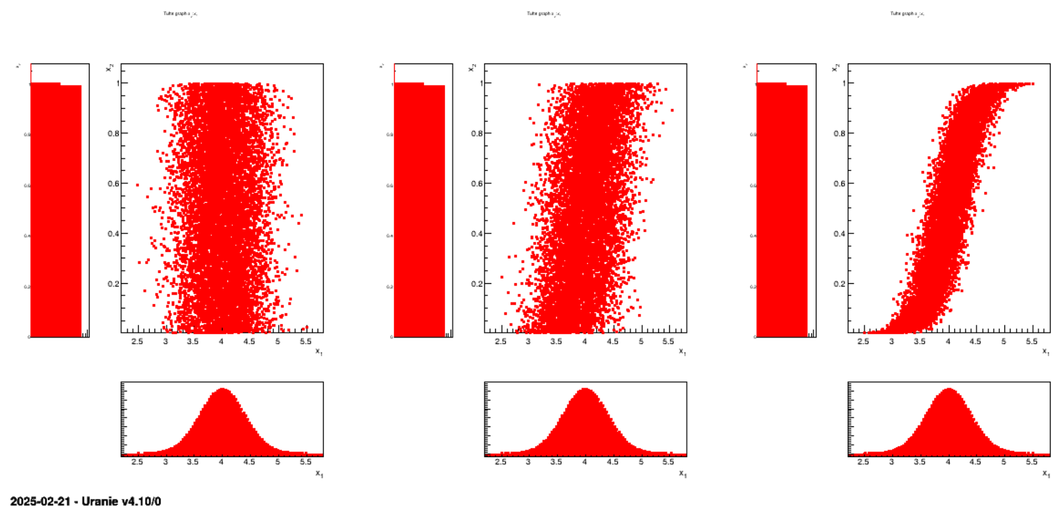


Figure III.4: Tufte plot of the design-of-experiments created using a normal and uniform distribution, with a LHS method with three correlation coefficient: 0, 0.45 and 0.9

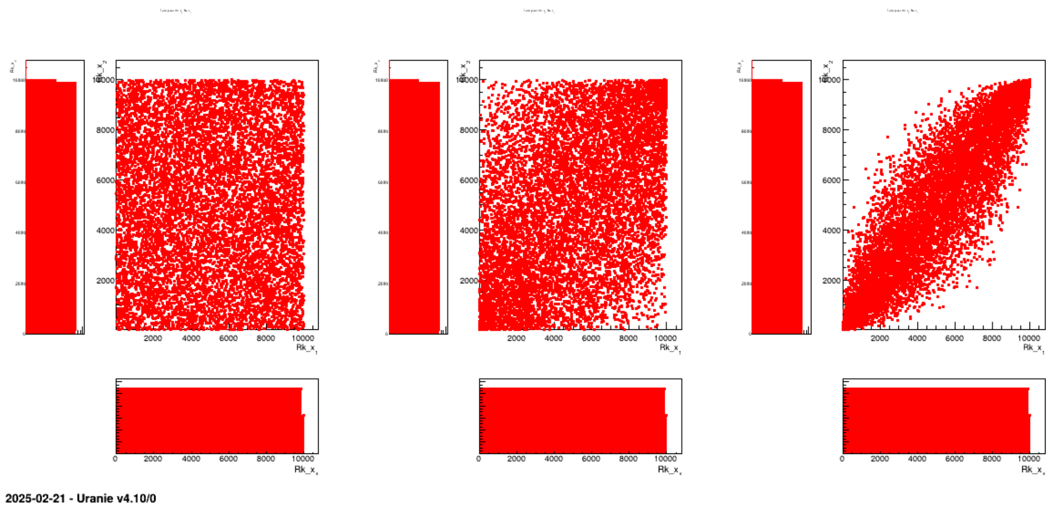


Figure III.5: Tufte plot of the rank of the design-of-experiments created using a normal and uniform distribution, with a LHS method with three correlation coefficient: 0, 0.45 and 0.9

Instead of provided one-by-one the correlation factors, one can provide the correlation matrix using the `setCorrelationMatrix` method of both `TSampling` and `TBasicSampling`. This method ask for a `TMatrixD` input and it will check that:

- the dimension is correct : (n_X, n_X).
- the coefficient are all in the range $[-1, 1]$
- the diagonal term are exactly at one (if not it sets them to this value).

III.3.1.1 Case of singular correlation matrix

In the case where the correlation matrix injected (at once or component by component) happen to be singular, the default method used to correlate variables will failed and complains in this way (taking the `TBasicSampling` as an example):

```
<URANIE::ERROR>
<URANIE::ERROR> *** URANIE ERROR ***
<URANIE::ERROR> *** File[${SOURCEDIR}/meTIER/sampler/souRCE/TBasicSampling.cxx] Line[427]
<URANIE::ERROR> TBasicSampling::generateCorrSample: the cholesky decomposition of the user ↩
    correlation matrix is not good.
<URANIE::ERROR>  Maybe you can try with SVD decomposition.
<URANIE::ERROR> *** END of URANIE ERROR ***
<URANIE::ERROR>
```

There is a workaround, provided by Uranie and heavily discussed in [30]. The idea in a nutshell is to remplace the Cholesky operation by a SVD one, to decompose the target correlation matrix. This can be done adding "svd" in the `Option_t *option` field of either the constructor of the class or the generator method call. Here is an example for the `TBasicSampling` class:

```
TBasicSampling *tbs = new TBasicSampling(tds,"lhs;svd",100); // example at constructor
//.. or ..
tbs->generateCorrSample("svd"); // example with generator function call
```

Here comes its equivalent piece of code for the `TSampling` class:

```
TSampling *ts = new TSampling(tds,"lhs;svd",100); // example at constructor
//.. or ..
ts->generateSample("svd"); // example with generator function call
```

In both cases, the use of this workaround should print this message to warn you about this.

```
<URANIE::INFO>
<URANIE::INFO> *** URANIE INFORMATION ***
<URANIE::INFO> *** File[${SOURCEDIR}/meTIER/sampler/souRCE/TBasicSampling.cxx] Line[371]
<URANIE::INFO> TBasicSampling::generateCorrSample: you've asked to use SVD instead of ↩
    Cholesky to decompose the correlation matrix.
<URANIE::INFO> *** END of URANIE INFORMATION ***
<URANIE::INFO>
```

An example of how to use this in real condition (with a proper singular correlation matrix) is shown in Section XIV.3.14.

III.3.2 The copula classes

Instead of using samplers with correlation matrix in the case of intricate variables, one can use classes relying on *Copula*, in order to describe the dependencies. These classes inherit from the TCopula class successively through the TSamplerStochastic and the TSampler classes. The idea of a copula is to define the interaction of variables using a parametric-function that can allow a broader range of entanglement than only using a correlation matrix (various shapes can be done). There are two kinds of copulas provided in the Uranie platform:

- **TArchimedian** with 4 pre-defined parametrisation: Ali-Mikhail-Haq (TAMHCopula), Clayton (TClaytonCopula), Frank (TFrankCopula) and Plackett (TPlackettCopula). These copulas, which depend only on the input variables and a parameter θ, are shown from Figure III.6 to Figure III.9 for different θ values along with the formula of the corresponding parametric function.

- **TElliptical:** it is a class to interface the TGaussian copula method.

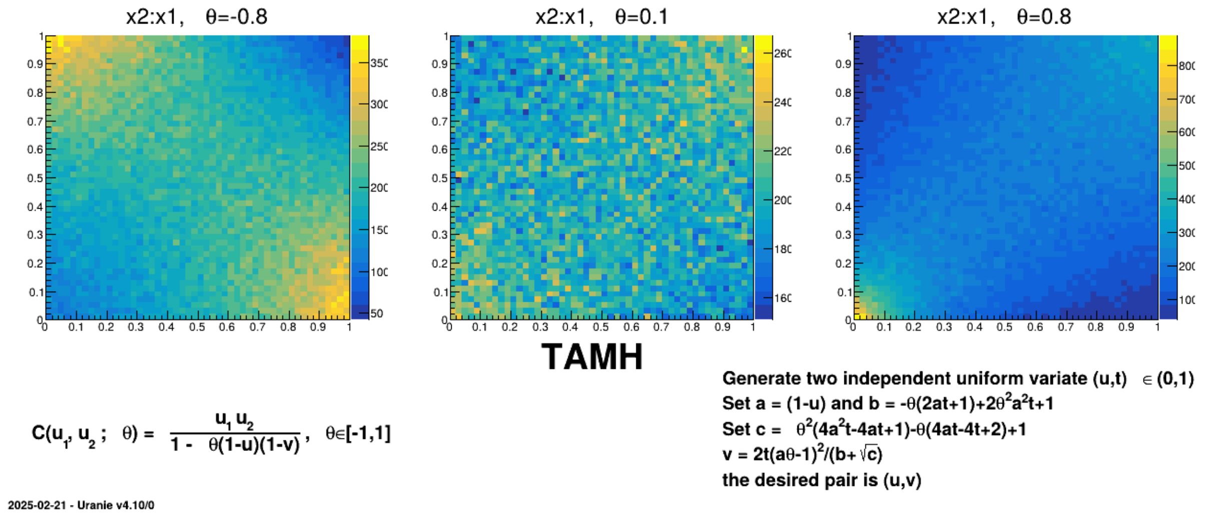


Figure III.6: Example of sampling done with half million points and two uniform attributes (from 0 to 1), using AMH copula and varying the parameter value.

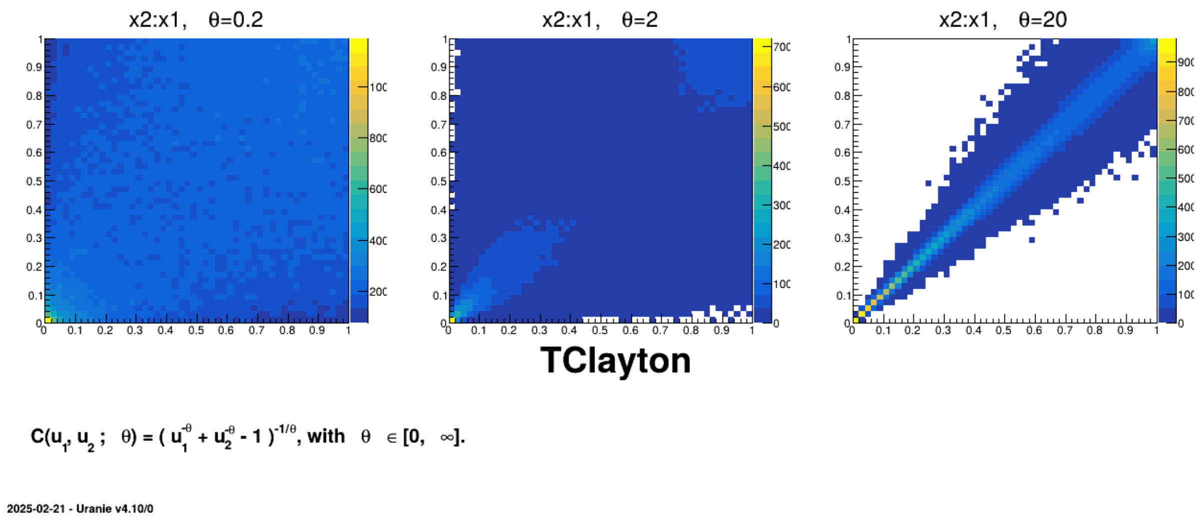


Figure III.7: Example of sampling done with half million points and two uniform attributes (from 0 to 1), using Clayton copula and varying the parameter value.

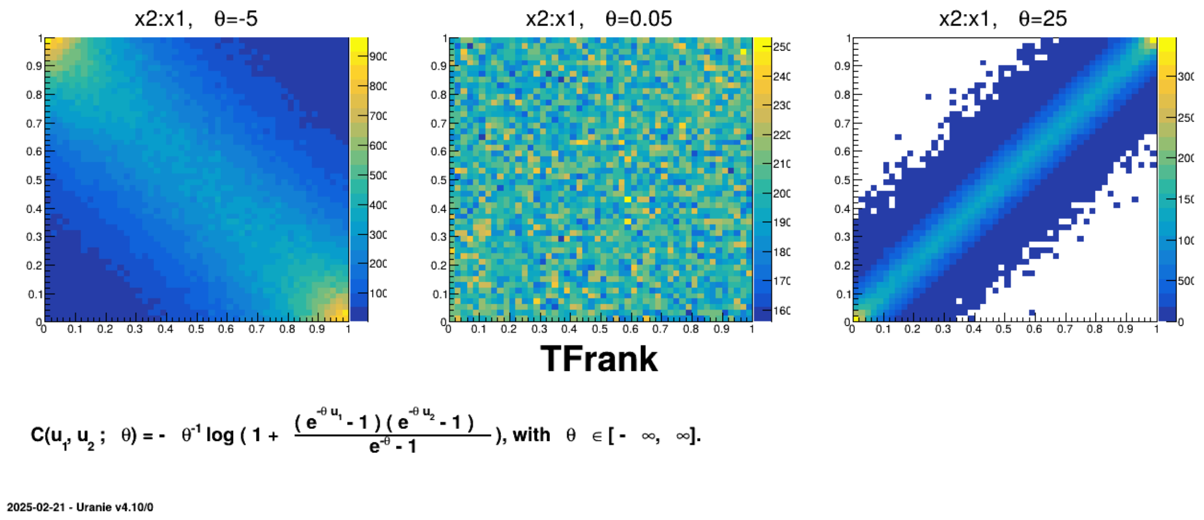
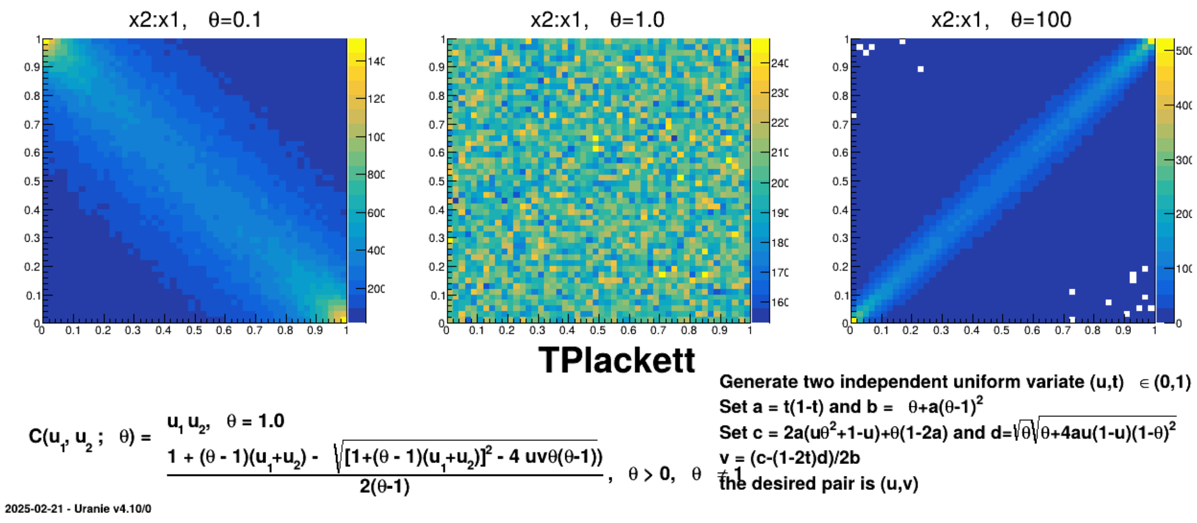


Figure III.8: Example of sampling done with half million points and two uniform attributes (from 0 to 1), using Frank copula and varying the parameter value.



$$C(u_1, u_2 ; \theta) = \frac{1 + (\theta - 1)(u_1+u_2) - \sqrt{[1+(\theta-1)(u_1+u_2)]^2 - 4 u v \theta(\theta-1)}}{2(\theta-1)}, \quad \theta > 0, \quad \theta \neq 1; \quad u_1 u_2, \quad \theta = 1.0$$

Figure III.9: Example of sampling done with half million points and two uniform attributes (from 0 to 1), using Plackett copula and varying the parameter value.

The following example shows how to create a copula and to use it in order to get a correlated sample.

```
TUniformDistribution *xunif = new TUniformDistribution("x1", 3., 4.);
TNormalDistribution  *xnorm = new TNormalDistribution("x2", 0.5, 1.5);

TDataServer * tds = new TDataServer("tdsSampling", "Demonstration Sampling");
tds->addAttribute(xunif);
tds->addAttribute(xnorm);

// Generate the sampling from the TDataServer object
TAMHCopula *tamh = new TAMHCopula(tds, 0.75, 1000);
tamh->generateSample();
tds->Draw("x2:x1","","colZ");
```

III.4 QMC method

The deterministic samplings can produce design-of-experiments with well defined properties, that can be very useful in specific cases such as:

- to cover at best the space of the input variables
- to explore the extreme cases
- to study combined or non-linearity effect

There are two kinds of quasi Monte-Carlo sampling methods implemented in Uranie: the regular ones and the sparse grid ones. On the first hand, the former can be generated using two different sequences:

1. Sequences of **Sobol** [7]
2. Sequences of **Halton** [8]

Figure III.10 shows a comparison of the design-of-experiments obtained with both sequences, along with the ones produced with a basic stochastic sampling, following the LHS and SRS "recipes", all when dealing with two uniform attributes. The coverage is clearly more regular in the case of quasi Monte-Carlo sequences which is the origin of their name: low-discrepancy sequences. There are plenty definitions for the notion of discrepancy (see litterature for them) but they all quantify how close the sequence is to a perfect equidistribution of points.

Warning
The **Halton** sequence has been designed initially to deal with uniform probability law. Extending their use to all probability laws, particularly to infinite-based, it is crucial to set at least lower-bound to these. The **Halton** sequence first value is indeed 0 and this means that going back from probability space to physical one, would imply $-\infty$ value if not properly bounded.
From version 4.3.0, the `TQMC` will complain about infinite-based law if lower-bounded and from version 4.6.0 it will be fully deprecated.

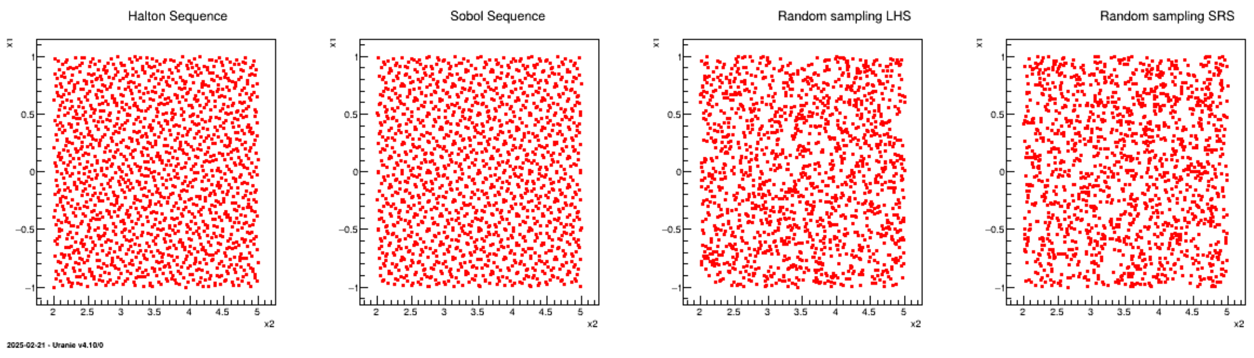


Figure III.10: Comparison of both quasi Monte-Carlo sequences with both LHS and SRS sampling when dealing with two uniform attributes.

On the other hand, the sparse grid sampling can be very useful for integration purposes and can be used in some of the meta-modelling definition, see, for instance, in Section V.3.1.2. In Uranie we can used the Petras algorithm [9] to produce these sparse grids, shown for different levels in Figure III.11, that can be compared to regular algorithms ones in Figure III.10 (in both cases, the problem is described with two uniform attributes).

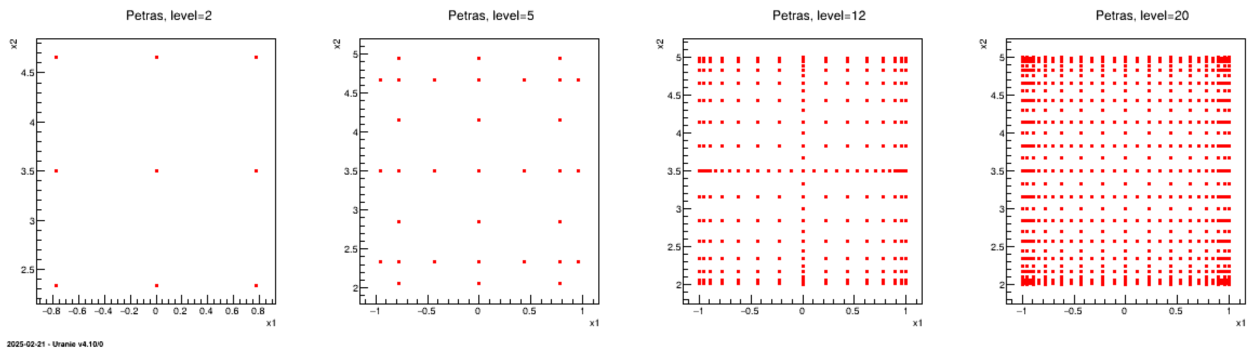


Figure III.11: Comparison of design-of-experiments made with Petras algorithm, using different level values, when dealing with two uniform attributes.

In the case of regular sequence, the selected sequence is specified with the second argument **option** of the class constructor `TQMC`:

```
TQMC(TDataServer *tds, Option_t *option, Int_t nCalcul );
```

First, a pointer to a `TDataServer` is constructed

```
TDataServer *tdsqmc = new TDataServer("tdsQMC", "Test for qMC");
tdsqmc->addAttribute( new TAttribute("x_{1}", -1.0 , 2.) );
tdsqmc->addAttribute( new TNormalDistribution("x_{2}",  0.0 , 3.) );
tdsqmc->addAttribute( new TAttribute("x_{3}",  1.0, 1.5) );
```

Then, a pointer to a TQMC object is constructed from a `TDataServer`, of the opted sequence among **("sobol"|"halton")**, with the wished size. At the end, the method `generateSample()` is applied

```
TQMC * qmc = new TQMC(tdsqmc, "sobol", 300);
qmc->generateSample();
```

III.5 The random fields

The random fields allow to take into account the spatial characteristics of a random variable.

A `TDataServer` is associated to a spatial random variable from $\mathbb{R}$, $\mathbb{R}^2$ or $\mathbb{R}^3$ in $\mathbb{R}$. Currently only one **Spectral** method, with two types of variograms (**Gaussian** and **Sine Cardinal**), is available in Uranie.

For instance, Uranie code is:

```
// Define the DataServer
TDataServer *tds = new TDataServer("TDSField", "Weight as Field");
tds->addAttribute( new URANIE::DataServer::TAttribute("x", 1, 51));
tds->addAttribute( new URANIE::DataServer::TAttribute("y", 1, 51));
tds->addAttribute( new URANIE::DataServer::TAttribute("Weight"));

// Gaussian Sampler Field
TSamplerField * tsf = new TSamplerField(tds); ❶[1]
tsf->SetScaleFactor(10.0);
tsf->SetRandomFunction(1000);
tsf->setVariogram("gauss");
tsf->SetVariance(1.0);

tsf->generateSample();❷[2]

// Graphic
TCanvas* canvas = new TCanvas();
canvas->Divide(1,2);
canvas->cd(1);   tds->Draw("Weight:y:x");
canvas->Modified();
canvas->Update();
canvas->cd(2);
tsf->Draw2D("x","y","tri2Z"); ❸[3]
```

❶ Allocation of a pointer of a random variable sampler with variables described in the `TDataServer` "tds" that is based on a "gaussian" variogram type. The constructor prototype is `TSamplerField(URANIE::DataServer: *tds, Option_t *option = "Gauss")`.

❷ The `generateSample()` method generates a sample which will be saved in the `TDataServer`.

❸ This step, along with the other drawing method few line earlier, gives the results shown in Figure III.12.

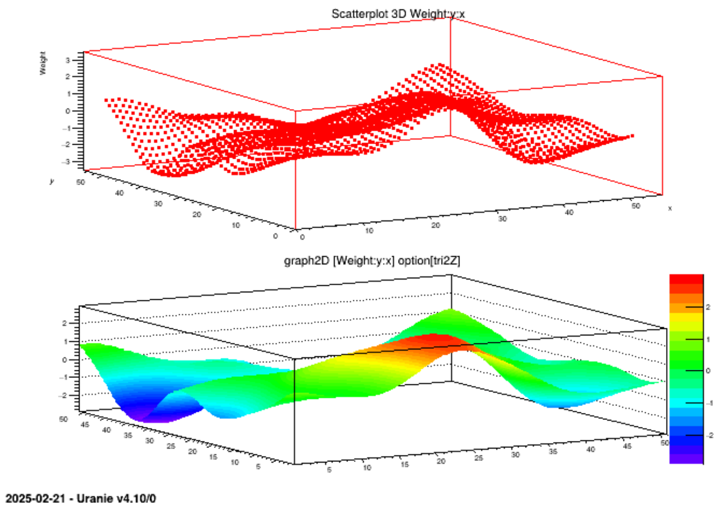


Figure III.12: Gaussian Random Field

It is possible to vary the value of the parameters used to construct the field, leading to different shapes. Examples of this are shown for the two implemented types of variograms, Gaussian in Figure III.13 and for Sine Cardinal in Figure III.14.

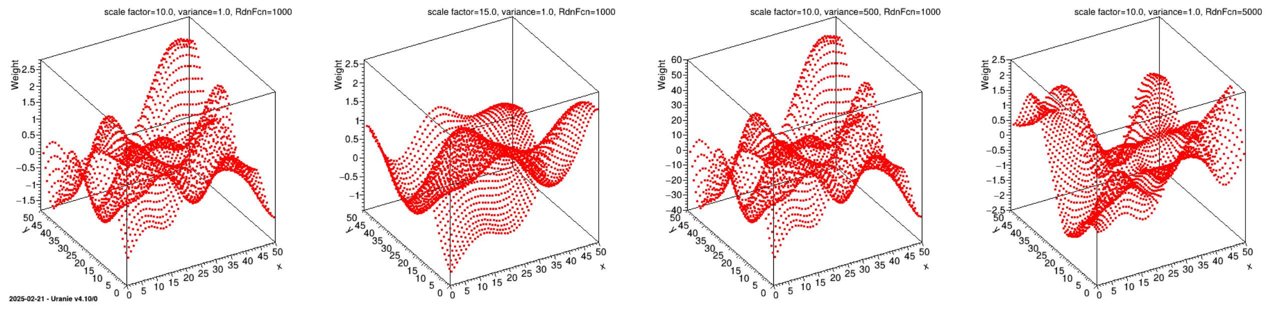


Figure III.13: Gaussian variograms. Several configurations (in terms of scale factor and variance parameters) are shown as well.

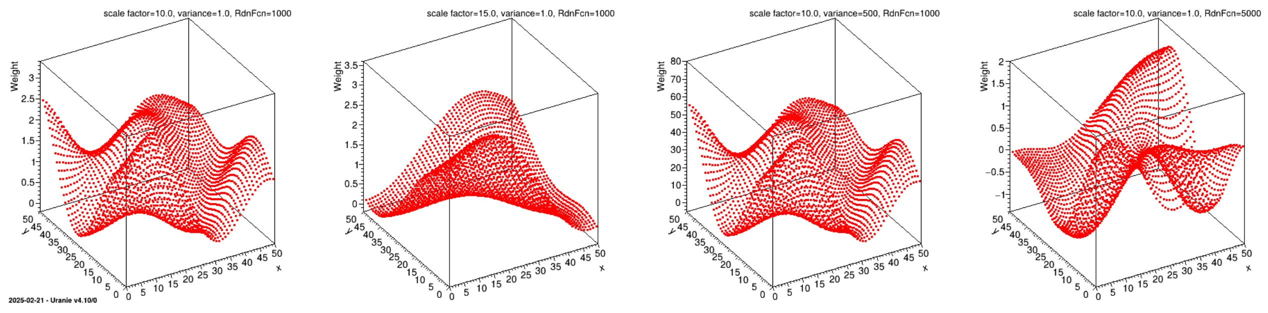


Figure III.14: Sine cardinal variograms. Several configurations (in terms of scale factor and variance parameters) are shown as well.

III.6 OAT Design

III.6.1 Introduction

The idea of the **One factor At a Time** (OAT) design-of-experiments is to observe the evolution of phenomena when one input factor is modified while all the others remain unchanged.

This design-of-experiments can be used to compute partial derivatives or sensitivities. For the latter application, it is interesting as it provides a very simple and inexpensive way to evaluate the sensitivity of a phenomena to its input factors. However, it might not be the best solution as it does not explore multi-factorial and non-linear effects.

III.6.2 OAT design in Uranie

In Uranie, each factor of the OAT design takes at least three values: a *nominal value*, a *lower value* (smaller than the nominal one) and an *upper value* (greater than the nominal one). When a factor is not "modified" it is set to its nominal value. This leads to a design of $2n+1$ experiments, where n is the number of factors, and ”+1” refers to a reference experiment, where no factor is modified.

Two classes can produce OAT designs:

- **`TOATSampling`:** simple, but limited in its functionality. It is used by some classes of Uranie.
- **`TOATDesign`:** an improved version of the previous one, more user oriented.

As **`TOATSampling`** will become deprecated in a future version of Uranie the present document will focus on the usage of **`TOATDesign`**.

III.6.3 `TOATDesign`

III.6.3.1 Construction of a simple OAT design-of-experiments

The easiest way to create an OAT design-of-experiments using `TOATDesign` is to proceed as follows:

1. create a dataserver with a list of attributes corresponding to the input factors;
2. set the default value of each attribute to the nominal value;

3. create a `TOATDesign` object, with the dataserver as first parameter;
4. define the **maximum range of variation** of the factors using the **`setRange`** function;
5. generate the OAT design-of-experiments using the **`generateSample`** function.

At this point, the dataserver contains an OAT design-of-experiments where each factor of interest is modified twice. Below is an example:

```
{
   // step 1
   TDataServer *tds = new TDataServer("tdsoat","Dataserver simple OAT design");
   tds->addAttribute(new TAttribute("x1"));
   tds->addAttribute(new TAttribute("x2"));

   // step 2
   tds->getAttribute("x1")->setDefaultValue(0.0);
   tds->getAttribute("x2")->setDefaultValue(10.0);

   // step 3
   TOATDesign *oatSampler = new TOATDesign(tds);

   // step 4
   Bool_t use_percentage = kTRUE;
   oatSampler->setRange("x1", 2.0);
   oatSampler->setRange("x2", 40.0, use_percentage);

   // step 5
   oatSampler->generateSample();

   // display
   tds->scan();
  }
```

This example produces:

```
****************************************************************************
*    Row   * tdsoat__n__iter * x1 * x2 * __nominal_set__ * __modified_att_ *
****************************************************************************
*        0 *               1 *  0 * 10 *               1 *              -1 *
*        1 *               2 * -1 * 10 *               1 *               1 *
*        2 *               3 *  1 * 10 *               1 *               1 *
*        3 *               4 *  0 *  8 *               1 *               2 *
*        4 *               5 *  0 * 12 *               1 *               2 *
****************************************************************************
```

In the example above, we have two input factors: x1 and x2. Their nominal values are respectively 0.0 and 10.0, and their maximum variation ranges are 2.0 and 40% of 10.0, i.e. 4.0.

Tip
To indicate that the range of "x2" is a percentage of its nominal value, we simply need to set the third parameter of the **setRange** function to TRUE. This parameter is set to FALSE by default, which means that the value of the range is considered to be "absolute".

The generated OAT design thus contains 5 experiments:

- the reference, where x1 and x2 are set to their nominal values;

- two variations of x1, where it equals -1.0 and 1.0 while x2 remains equal to 10.0;
- two variations of x2, where it equals 8.0 and 12.0 while x1 is set back to 0.0;

It also contains two new attributes, automatically added by Uranie:

- **__nominal_set__:** identifies which set of nominal values is used as a reference. In this case, we have only one set, thus __nominal_set__'s value remains equal to 1. This is further discussed in Section III.6.3.5
- **__modified_att__:** identifies which factor has been modified in the current experiment. The value is the index of the corresponding attribute in the dataserver. A value equal to -1 means that all the factors have their nominal values.

Warning
The index of an attribute in the dataserver can be different from the one printed by the scan function, or inside an output file. To be certain to retrieve the correct number, always use the **`TDataServer::getAttributeIndex`** function.

III.6.3.2 Some options

When creating a new OAT sampler object, the following options are available:

- **sampling mode**: determines how the modified factor's new values are selected inside the interval $\left[nominal - \frac{range}{2}; nomina\right.$
It can be either:
  - **regular:** an even number of new values and the nominal value are regularly distributed along the interval.
  - **lhs**, **srs** or **random:** the new values are randomly chosen inside the interval using a Latin Hypercube Sampling ("lhs") or a Standard Random Sampling ("srs" and "random") (cf. Figure III.3). The sample's distribution is given by the type of the attribute representing the factor (cf. Section II.2.5).
- **number of modifications:** the number of new values taken by a modified factor. It must be greater or equal to 2. If the sampling mode is "regular" and the given number M is odd, the actual number of modifications will be $M-1$.

In the example of Section III.6.3.1, we used the default options for the OAT sampler, namely:

- sampling mode: regular;
- number of modifications: 2.

In the next examples, we present the results of choosing different options.

III.6.3.3 Regular mode

In the example of Section III.6.3.1, if we change the "step 3" to:

```
TOATDesign oatSampler(tds, "regular", 4);
```

the OAT design becomes:

```
********************************************************************************
*    Row   * tdsoat__n__iter * x1.x * x2. * __nominal_set__ * __modified_att_ *
********************************************************************************
*        0 *               1 *    0 *  10 *               1 *              -1 *
*        1 *               2 *   -1 *  10 *               1 *               1 *
*        2 *               3 *    1 *  10 *               1 *               1 *
*        3 *               4 * -0.5 *  10 *               1 *               1 *
*        4 *               5 *  0.5 *  10 *               1 *               1 *
*        5 *               6 *    0 *   8 *               1 *               2 *
*        6 *               7 *    0 *  12 *               1 *               2 *
*        7 *               8 *    0 *   9 *               1 *               2 *
*        8 *               9 *    0 *  11 *               1 *               2 *
********************************************************************************
```

Each factor is modified 4 times, and its values are regularly spaced over the interval of variation (see Section XIV.3.6 for complete code).

III.6.3.4 Random mode

In order to have randomly distributed values over the interval, the example's code needs further modifications.

To produce a random sampling, the attributes representing the factors must belong to the TStochasticAttribute family (cf. Section II.2.5). We thus need to modify the "step 1" of example of Section III.6.3.1 to:

```
// step 1
TDataServer *tds = new TDataServer("tds","Data server for simple OAT design");
tds->addAttribute(new TUniformDistribution("x1", -5.0, 5.0));
tds->addAttribute(new TNormalDistribution("x2", 11.0, 1.0));
```

Tip
There is no *a priori* relationship between the distribution of the attribute and the nominal value and range of the factor it represents. However, a good practice is to insure that the probability density over the whole factor's range is never null.

The "step 2" of the example of Section III.6.3.1 does not need to be modified. The "step 3", on the other hand, becomes:

```
// step 3
TOATDesign oatSampler(tds, "lhs", 1000);
```

By choosing the "lhs" mode, we ask for a random sampling over the range of the factor (defined in "step 4"). Here, we also ask for 1000 [1] modifications of each factor.

The rest of the script remains unchanged. We modify the "//display" section in order to visualise histograms of the sampling, instead of a long list of numbers:

```
 // display
TCanvas c("can");
c.Divide(2,1);
c.cd(1);
tds->draw("x1","__modified_att__ == 1");
c.cd(2);
tds->draw("x2","__modified_att__ == 2");
```

[1] This is a ridiculously high number for an OAT design whose aim is, precisely, to provide a small and simple design-of-experiments. We do this only to be able to visualise nice histograms !

Tip
The two calls to `draw` are an illustration of the use of the "__modified_att__" attribute. Here, it allows to filter out the data, by keeping only the experiments where the interesting factor is modified.

The resulting histograms are shown in figure Figure III.15. The left histogram shows the distribution of data for the "x1" attribute (uniform distribution) and the right, for the "x2" attribute (gaussian distribution). The latter shows how the gaussian distribution is truncated by the choice of the nominal value and the range (the complete code can be found in Section XIV.3.7).

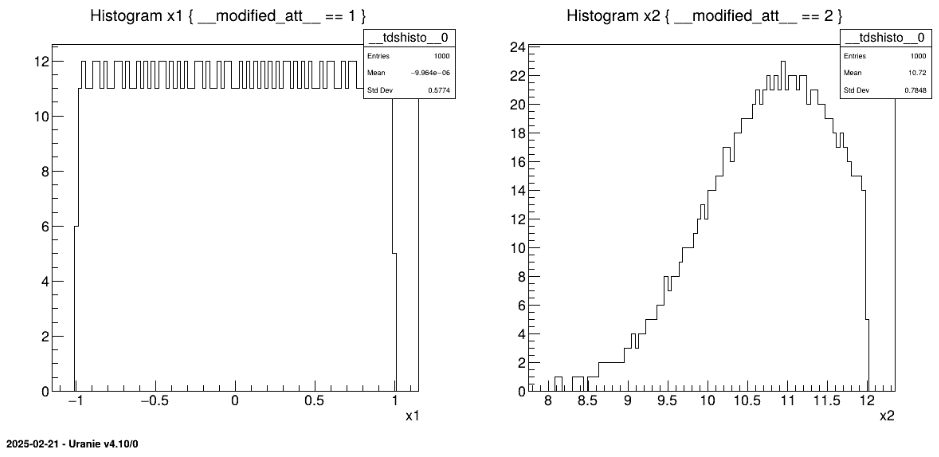


Figure III.15: Random values for OAT design

III.6.3.5 Multiple sets of nominal values

If the factors have more than one possible nominal value, the OAT sampler can automatically build a design-of-experiments for each set of nominal values stored in the dataserver.

This information can be set manually, but an easier way is to write it inside a "Salome table" file (cf. Section II.3 for a description of the format). Below is a simple example of such a file:

```
#FILE_NAME: myNominalValues.dat
#COLUMN_NAMES: x1 | x2

0.0   10.0
5.0    3.0
-5.0   17.0
```

Still starting from the script of the example of Section III.6.3.1, the only step requiring modifications is the first one:

```
// step 1
TDataServer *tds = new TDataServer("tds","Data server for simple OAT design");
tds->fileDataRead("myNominalValues.dat");
```

The "step 2" is now useless and must be removed. The nominal values of the factors will be automatically loaded from the dataserver.

Now if we run the modified script, the result is:

```
*********************************************************************************
*    Row   * tdsoat__n__iter *   x1 *   x2 * __nominal_set__ * __modified_att_ *
*********************************************************************************
*        0 *               1 *    0 *   10 *               1 *              -1 *
*        1 *               2 *   -1 *   10 *               1 *               0 *
*        2 *               3 *    1 *   10 *               1 *               0 *
*        3 *               4 *    0 *    8 *               1 *               1 *
*        4 *               5 *    0 *   12 *               1 *               1 *
*        5 *               6 *    5 *    3 *               2 *              -1 *
*        6 *               7 *    4 *    3 *               2 *               0 *
*        7 *               8 *    6 *    3 *               2 *               0 *
*        8 *               9 *    5 *  2.4 *               2 *               1 *
*        9 *              10 *    5 *  3.6 *               2 *               1 *
*       10 *              11 *   -5 *   17 *               3 *              -1 *
*       11 *              12 *   -6 *   17 *               3 *               0 *
*       12 *              13 *   -4 *   17 *               3 *               0 *
*       13 *              14 *   -5 * 13.6 *               3 *               1 *
*       14 *              15 *   -5 * 20.4 *               3 *               1 *
*********************************************************************************
```

The generated design contains $3 \times (2n+1)$ experiments. The attribute "__nominal_set__" now varies from 1 to 3, indicating which set of nominal value is taken as reference. The complete code can be found in Section XIV.3.8.

III.6.3.6 Multiple ranges

We have seen that it is possible to interpret the range of variation as a percentage of the nominal value. This is often enough to adapt the range to very different nominal values. However, in some contexts it can be useful to *really* modify the range of a factor.

This information can also be read from a data file. For example, our previous file can be modified to add a new attribute representing the range of one of the factors:

```
#FILE_NAME: myNominalValues.dat
#COLUMN_NAMES: x1 | x2 | rx1

0.0   10.0   2.0
5.0    3.0   0.4
-5.0   17.0   6.0
```

Tip
The name of the attribute and the order in which it appears in the file is meaningless. Actually, any attribute can be considered as a range.

The modifications of "step 1" and the removal of "step 2" in the example of Section III.6.3.5 are still valid, while "step 4" must be modified in order to tell the OAT sampler that the range of "x1" is represented by another attribute:

```
// step 4
Bool_t use_percentage = kTRUE;
oatSampler.setRange("x1", "rx1");
oatSampler.setRange("x2", 40.0, use_percentage);
```

Now, for each new set of nominal values, the value of "rx1" will become the range of "x1".

The result of the modified script (which can be found in Section XIV.3.8) is:

```
*********************************************************************************
*    Row   * tds__n__iter *   x1 *   x2 *  rx1 * __nominal_set__ * __modified_att_ *
*********************************************************************************
*        0 *            1 *    0 *   10 *    2 *               1 *              -1 *
*        1 *            2 *   -1 *   10 *    2 *               1 *               0 *
*        2 *            3 *    1 *   10 *    2 *               1 *               0 *
*        3 *            4 *    0 *    8 *    2 *               1 *               1 *
*        4 *            5 *    0 *   12 *    2 *               1 *               1 *
*        5 *            6 *    5 *    3 *  0.4 *               2 *              -1 *
*        6 *            7 *  4.8 *    3 *  0.4 *               2 *               0 *
*        7 *            8 *  5.2 *    3 *  0.4 *               2 *               0 *
*        8 *            9 *    5 *  2.4 *  0.4 *               2 *               1 *
*        9 *           10 *    5 *  3.6 *  0.4 *               2 *               1 *
*       10 *           11 *   -5 *   17 *    6 *               3 *              -1 *
*       11 *           12 *   -8 *   17 *    6 *               3 *               0 *
*       12 *           13 *   -2 *   17 *    6 *               3 *               0 *
*       13 *           14 *   -5 * 13.6 *    6 *               3 *               1 *
*       14 *           15 *   -5 * 20.4 *    6 *               3 *               1 *
*********************************************************************************
```

If we compare this design to the previous one, we can see that the column for "x2" is unchanged, while "x1" is modified according to the value of "rx1". We can also note that "rx1" is never modified (in the OAT way). This is because no call to the function `setRange` was done for it. **Only the attributes with an associated range are modified by the sampling procedure**.

III.6.3.7 Remarks

To finish this description of the OAT sampler of Uranie, here are some general remarks to answer frequently asked questions or to inform the user about the evolution of the class.

- It is not possible to use random modes when data are loaded from a file.
- The value of a range can always be interpreted as percentage of the nominal value, even if the range is read from the dataserver. Please refer to the developer documentation of the function `URANIE::Sampler::TOATDesign::setRa` for details.

III.7 The Vectorial Quantification method

This method is called when instead of having a list of input parameters (in terms of stochastic distribution) that one would like to transform into a design-of-experiments, the user has a dataset, made out of a very large number of points. In this case, it is possible, using the `TNeuralGas` class, to create sub-sample of points that would be representative of the complete provided-set, based on a NeuralGas algorithm. This might be useful in order to test the output of long and complicated codes or computations without leaving aside a possible area of the input parameter values.

Figure III.16 shows the effect of the reduction of the sample in the simple case of a two-dimensional plane, when considering the "geyser.dat" file and its sub-sample of 50 points.

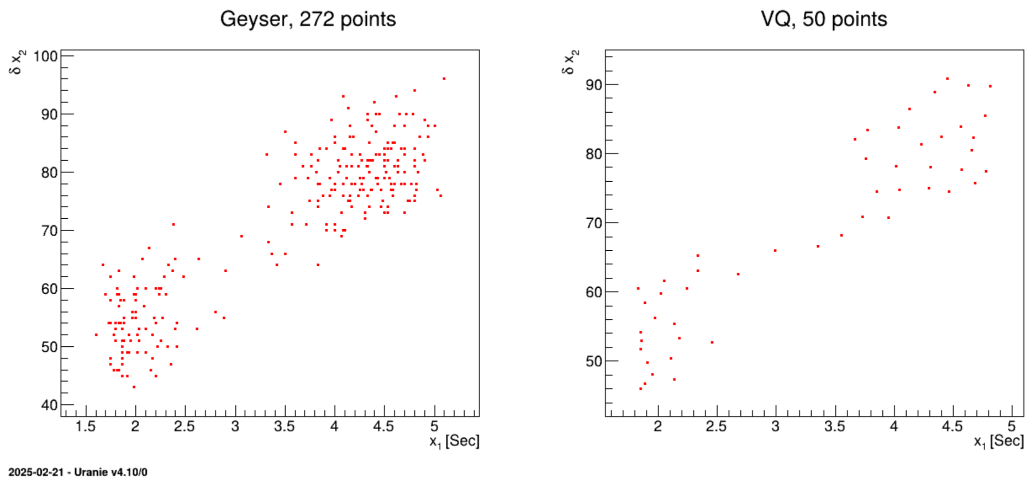


Figure III.16: Example of a dataset reduction (the geyser one) using the NeuralGas algorithm, to go from 272 points (left) to 50 one (right)

Here is an example of how to use the neuralgas algorithm to reduce a database.

```
TCanvas *c = new TCanvas("Can","Can",10,32,1300,600); ❶[1]
c->Divide(2,1);

TDataServer *tdsGeyser = new TDataServer("tdsgeyser", "Neural Gas for Geyser");
tdsGeyser->fileDataRead("geyser.dat");

TNeuralGas * tvq = new TNeuralGas(tdsGeyser,"", 50); ❷[2]
tvq->setDrawProgressBar(kFALSE);
TDataServer *tdsng = tvq->getSubSample("loop=20"); ❸[3]

c->cd(1);
tdsGeyser->draw("x2:x1");
((TH2F*)gPad->GetPrimitive("__tdshisto__0"))->SetTitle("Geyser, 272 points"); ❹[4]

c->cd(2);
tdsng->draw("x2:x1");
((TH2F*)gPad->GetPrimitive("__tdshisto__0"))->SetTitle("VQ, 50 points");
```

Construction of the plot

❶ Creation of `TCanvas` subdivided into two pieces to compare results.

❷ Construction of the neuralgas object from the provided database

❸ Get the sub-sample as a new dataserver, after looping 20 the algorithm

❹ Access the latest histogram drawn on current pad, to change its title

Chapter IV

The Launcher module

IV.1 Introduction

This chapter introduces the launcher module whose purpose is to get a complete response surface from a design-of-experiments. This can be done using the relauncher module as well (see Chapter VIII) whose goals are the same. The main difference lies in the way they both are dealing with parallelisation of computation: the launcher module relies on an automatic detection of a possible cluster Section IV.4 whereas the relauncher is designed for a local parallelisation (on the current machine with different technologies, as explained in Section VIII.4).

IV.1.1 Presentation

We present in this chapter the features of Uranie, related to the assessment of a design-of-experiments (matrix of "X") to construct the output variables (matrix of "Y"). Uranie can manage two kinds of computations: analytic functions and external codes.

In the first case of an *analytic computation* (Section IV.2), an analytic formula is used to compute outputs from inputs.

In the second case, Uranie calls an *external computational code* (Section IV.3). The definition of the code must contain:

1. the input files where the **variables** under study stand;
2. output files containing the **computational results** .

Input and output files can have multiple data storage formats (see Section IV.3.1).

Moreover, the code assessment can be either sequential or distributed (Section IV.4). In both cases, the same Uranie macro will be used.

IV.1.2 Overview of a simple case

To illustrate these features, let us consider the definition of the **flowrate** problem, that we will implement as an analytic function (see Section IV.1.2.2) as well as an external code (see Section IV.1.2.3).

IV.1.2.1 Presentation of the problem

This problem is handled in [39], page 35, with the mathematical formula to be treated.

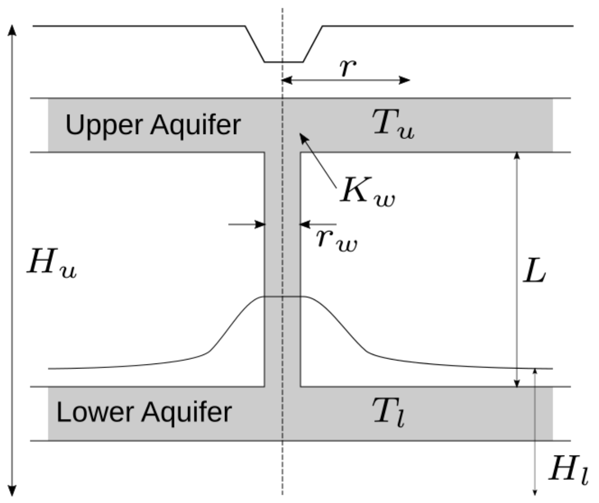


Figure IV.1: Sketch of the flowrate problem and its variables[2].

$$y = f(x) = \frac{2\pi T_u (H_u - H_l)}{\ln(\frac{r}{r_\omega})\left[1 + \frac{2LT_u}{\ln(\frac{r}{r_\omega}) r_\omega^2 K_\omega} + \frac{T_u}{T_l}\right]}$$

EQUATION IV.1: Flowrate function

The eight parameters shown both in Figure IV.1 and in equation Equation IV.1 are introduced below by describing their meaning along with their underlying hypothesis, bearing in mind that it is common, in the Uranie context, to remind in the statistical approach.

1. $r_\omega \in [0.05, 0.15]$ (m): radius of borehole
2. $r \in [100, 50\ 000]$ (m): radius of influence
3. $T_u \in [63\ 070, 115\ 600]$ $(m^2/year)$: Transmitivity of the superior layer of water
4. $T_l \in [63.1, 116]$ $(m^2/year)$: Transmitivity of the inferior layer of water
5. $H_u \in [990, 1\ 110]$ (m): Potentiometric "head" of the superior layer of water
6. $H_l \in [700, 820]$ (m): Potentiometric "head" of the inferior layer of water
7. $L \in [1\ 120, 1\ 680]$ (m): length of borehole
8. $K_\omega \in [9\ 855, 12\ 045]$ (m): hydraulic conductivity of borehole

This example has been treated by several authors in the dedicated literature, for instance in [40].

IV.1.2.2 Case of an analytic function

We will focus on the **flowrate** benchmark (cf Section IV.1.2.1) to present the use of Uranie with an analytic function. In this case, the analytic function must be written with "C" format according to the following prototype:

```
void myFunction (Double_t *param, Double_t *res)
```

It is the classical prototype of ROOT for these objects TF1, TF2 and TF3. The **flowrate** function is written in the macro file "UserFunctions.C" which can be found in ${URANIESYS}/share/uranie/macros and which looks like this:

```
#include "TMath.h"

void flowrateModel(double *x, Double_t *y)
{
  Double_t drw = x[0], dr  = x[1];
  Double_t dtu = x[2], dtl = x[3];
  Double_t dhu = x[4], dhl = x[5];
  Double_t dl  = x[6], dkw = x[7];

  Double_t dnum = 2.0 * TMath::Pi() * dtu * ( dhu -dhl);
  Double_t dlnronrw = TMath::Log( dr / drw);
  Double_t dden = dlnronrw * ( 1.0 +  ( 2.0 * dl * dtu ) / ( dlnronrw * drw * drw * dkw) + ↩
      dtu / dtl );

  y[0] = dnum / dden;
}
```

IV.1.2.3 Case of an external code

It is assumed that the code can be executed by the user; we will deal with the setup of the code on the machines the user will use. Moreover, the environment variables PATH and LD_LIBRARY_PATH must be properly set up to allow the execution of ROOT and/or Uranie and/or any commands needed for the execution of the computational code.

Here we do not have an analytic function like described above anymore, but a computational code with its input files, with output variables being stored in ASCII files; these will be called *output files*.

We use the **flowrate** benchmark (cf Section IV.1.2.1) as this computational code for this example. Users must give to the code the appropriate input files. The binary corresponding to the benchmark, also named **flowrate**, is available in the bin directory of Uranie installation directory. Users can see options of **flowrate** by executing the command **flowrate -h**.

```
flowrate -h
Usage: flowrate [-d] [-s] [-k|-f|-r|-kf|-ff] [-h|-?] [file]
-d:   debug mode
-s:   silent mode
-k:   input file with keys  [flowrate_input_with_keys.in]
-f:   input file with flags [flowrate_input_with_flags.in]
-r:   input file with only values in rows [flowrate_input_with_values_rows.in]
-kf:  input file with keys with failure mode [flowrate_input_with_keys.in]
-ff:  input file with flags with failure mode [flowrate_input_with_flags.in]
-h,-?:   this help message
```

- **-d** option allows to give some intermediate values to users while executing **flowrate**, such as values read in input files, or to show steps or information in execution (opening of input files, number of lines of these input files...).
- **-s** silent mode.

- **-k** and **-kf** options allow to simulate the launch of **flowrate** with data files of "key=value" format (see Section IV.1.2.3.1.1). The **-k** simply executes the flowrate code. The **-kf** allows to determine a set of values for which **flowrate** is considered in failure. If user does not specify an input file, **flowrate** will look for both options the input file *flowrate_input_file_with_keys.in* whom "key=value" format is described below.
- In the same way, **-f** and **-ff** options allow to simulate the launch of **flowrate** code with files with flags (see Section IV.1.2.3.1.2). When **-f** allows a simple execution of **flowrate**, **-ff** allows to simulate a code failure for a specific set of data. If the user does not specify an input file, **flowrate** will look for the input file *flowrate_input_file_with_flags.in* whom "flags" format is described below.
- **-r** option allows to simulate the launch of **flowrate** code with files with values in rows (see Section IV.1.2.3.1.3). The default input file is *flowrate_input_with_values_rows.in* whom "values in rows" format is described below.

There are eight mandatory input variables (r_ω, r, T_u, T_l, H_u, H_l, L, K_ω) described in (Section IV.1.2.1). User can also defines three optional variables **chu**, **chl** and **cr** which allow to calculate a variable called **d** which is written in output files. However, these three values have an incidence on the execution of **flowrate** only with **-kf** and **-ff** options. The formula used to calculate **d** is the following:

```
d=(Hu - chu)*( Hu - chu) +  (Hl - chl)*( Hl  - chl)
```

This equality means **d** is the squared radius of the circle *C* of centre (**chu**,**chl**).

If **-kf** or **-ff** options are used, **flowrate** tests if

```
d < cr
```

i.e. we consider couples of values (**hu**,**hl**) which are inside the circle *C* as a set of values where **flowrate** fails. If this inequality is verified, **flowrate** crashes and no output file is produced. Without these options (**-kf** and **-ff**), this inequality is not checked.

IV.1.2.3.1 Flowrate input files

IV.1.2.3.1.1 Input files with "key=value" format

In this section, we describe the file *flowrate_input_with_keys.in*. Executing **flowrate** with options **-k** and **-kf** without specifying any input file, means that *flowrate_input_with_keys.in* is taken as a default entry. If the user creates "*.in" input files, the hereafter format must be kept, with the same variables names (r_ω, r, T_u, T_l, H_u, H_l, L, K_ω).

```
#
#
# INPUT FILE with KEYS for the "FLOWREATE" code
# \date   2008-04-22 12:53:35
#


date = 123456 ;


#########################
##
##  exclude points
##
chu = 1050;
chl = 770;
cr  = 1100;
##
```

```
#########################

#########################
##
##  parameters : 8
##
Rw = 0.0500 ;
R = 33366.67 ;
Tu = 63070.0 ;
Tl = 116.00 ;
Hu = 1110.00 ;
Hl = 768.57;
L = 1200.0 ;
Kw = 11732.14 ;
##
#########################


#########################
##
## to simulate CPU time
##
## normal         1 :
## min     10000000 : 1.160u 0.000s 0:01.16 100.0%
## max    100000000 : 11.600u 0.010s 0:11.61 100.0%
##
nLoop = 1;
##
#########################

end = 6;
```

When using this kind of input files, Uranie is able to make the parameters values vary so as to launch the code multiple times with different values. Note that all the variables of this file are not modified: the **date** and **nLoop** variables will remain unchanged during the study.

IV.1.2.3.1.2 Input files with flag format

In this section, we describe the file *flowrate_input_with_flags.in*. Executing **flowrate** with options **-f** and **-ff** without specifying any input file, means that *flowrate_input_with_flags.in* is taken as a default entry. In the same way as for files with "key=value" format, if the user wants to create a "*.in" input files with "flag" format, the same format has to be kept as for the file *flowrate_input_with_flags.in*.

On the contrary of the "key=value" format, changes are necessary for Uranie to be able to modify the values. For example, if one has the following input file:

```
#
# INPUT FILE with FLAG for the "FLOWREATE" code
# \date   2008-04-22 12:55:17
#

new Implicit_Steady_State sch {
    frottement_paroi { 0.071623 19712.541454 }       // values of Rw and R
    tinit               0.0
    tmax                1000000.
    nb_pas_dt_max       1500
    dt_min              1051.972855                  // value of Hu
    dt_max              805.249178                   // value of Hl
    facsec              1000000.
    kW                  11401.611060                    // value of Kw
    information_Tu      Champ_Uniforme 1        85927.853162    // value of Tu
    information_Tl      Champ_Uniforme 1        85.803614       // value of Tl
    information_L {
        precision  1162.689830                          // value of L
    }
    convergence {
        criterion relative_max_du_dt
        precision  1.e-6
    }

    stop_criterium {
        ch_abcsissa_hu 1050
        ch_ordinate_hl 770
        c_radius 1100
    }

    Solveur Newton3 {
        max_iter_matrice        1
        max_iter_implicite      1
        date                    5654321
        seuil_convg_implicite   1.e-6
        assemblage_implicite    10
        solveur_lineaire        BiCGS
            preconditionneur    ILU
            seuil_resol_implicite 1.e-5
    }
}
```

and one wants to change some values, then one shall replace these values with "flags", hinted with special characters. For example one can replace the values with their names surrounded by "@":

```
#
# INPUT FILE with FLAG for the "FLOWREATE" code
# \date   2008-04-22 12:55:17
#

new Implicit_Steady_State sch {
    frottement_paroi { @Rw@ @R@ }
    tinit               0.0
    tmax                1000000.
    nb_pas_dt_max       1500
    dt_min              @Hu@
    dt_max              @Hl@
    facsec              1000000.
    kW                  @Kw@
    information_Tu      Champ_Uniforme 1        @Tu@
    information_Tl      Champ_Uniforme 1        @Tl@
    information_L {
        precision  @L@
    }
    convergence {
        criterion relative_max_du_dt
        precision  @Rw@
    }

    stop_criterium {
        ch_abcsissa_hu 1050
        ch_ordinate_hl 770
        c_radius 1100
    }
    Solveur Newton3 {
        max_iter_matrice        1
        max_iter_implicite      1
        date                    5654321
        seuil_convg_implicite   1.e-6
        assemblage_implicite    10
        solveur_lineaire        BiCGS
            preconditionneur    ILU
            seuil_resol_implicite 1.e-5
    }
}
```

This file is not directly (without Uranie) exploitable by the **flowrate** code anymore (contrary to the file *flowrate_input_with_keys.* because flags are not numerical values. Therefore, the flags we have in the modified file, such as *@Rw@* for the variable *rw* or *@R@* for the variable *r*, have to be replaced by numerical values before it becomes usable.

IV.1.2.3.1.3 Input files with values in rows

We describe here the shape of the default file *flowrate_input_with_values_rows.in* searched by **flowrate** with the **-r** option. Typically, the expected input file looks like the following: eight values which are the values of the eight variables defined above (r_ω, r, T_u, T_l, H_u, H_l, L, K_ω). To define the three optional variables, users must add three values just after other values. In the contrary, the three default values are respectively **chu=1050**, **chl=770** and **cr=1100**.

```
0.0500 33366.67 63070.0 116.00 1110.00 768.57 1200.0 11732.14
```

IV.1.2.3.2 Flowrate output files

The **flowrate** code generates two output files:

- *_output_flowrate_withRow_.dat*: file with "column" format and with a header containing the names of the cost variable and the **d** variable. Then, the values of the cost variable and the **d** variable are displayed.

  ```
  #COLUMN_NAMES: yhat | d

  6.757218e+01  4.092561e+03
  ```

- *_output_flowrate_withKey_.dat*: file with "key=value" format without an header;

  ```
  yhat = 6.757218e+01;
  d = 4.092561e+03;
  ```

IV.2 Analytic function

The analytic functions compatible with Uranie must follow the prototype:

```
void MyFunction(Double_t *x, Double_t *y)
```

where:

- `MyFunction` is the name of the function;
- `x` is an array of `Double_t` which represents inputs;

- y is an array of Double_t which represents outputs or targets.

So, this prototype allows us to execute any function of $\mathbb{R}^{nX}$ on $\mathbb{R}^{nY}$.

The example below is the content of the file UserFunctions.C in which the definition of the flowrate function seen in Section IV.1.2.1 has been copied:

```
#include "TMath.h"

void flowrateModel(double *x, Double_t *y)
{
  Double_t drw = x[0], dr  = x[1];
  Double_t dtu = x[2], dtl = x[3];
  Double_t dhu = x[4], dhl = x[5];
  Double_t dl  = x[6], dkw = x[7];

  Double_t dnum = 2.0 * TMath::Pi() * dtu * ( dhu -dhl);
  Double_t dlnronrw = TMath::Log( dr / drw);
  Double_t dden = dlnronrw * ( 1.0 +  ( 2.0 * dl * dtu ) / ( dlnronrw * drw * drw * dkw) +  ↩
      dtu / dtl );

  y[0] = dnum / dden;
}
```

In a Uranie script, after having defined the TDataServer object and created a sample, we load the previously defined function and create a TLauncherFunction object with its name as second argument, as shown below:

```
// Create a TDataServer
TDataServer * tds = new TDataServer("tdsFlowrate","TDS for flowrate");   ❶[1]
// Add the eight attributes of the study with uniform laws
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));          ❷[2]
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

// Generate the sampling from the TDataServer
TSampling *sampling = new TSampling(tds, "lhs", 1000);                     ❸[3]
sampling->generateSample();

// Load the function in the UserFunction macros file
gROOT->LoadMacro("UserFunctions.C");                                       ❹[4]

// Create a TLauncherFunction from a TDataServer and an analytic function
TString FuncName="flowrateModel";
TLauncherFunction * tlf = new TLauncherFunction(tds, FuncName,"",FuncName);      ❺[5]

// Define whether a progress bar should be drawn
tlf->setDrawProgressBar(kFALSE);                                           ❻[6]

// Evaluate the function on all the design-of-experiments
tlf->run();                                                                ❼[7]

tds->drawProfile("flowrateModel:rw");
```

Evaluate an analytic function

1. Create a `TDataServer`;
2. Add attributes to the `TDataServer`;
3. Create and generate a sample;
4. Load the file `UserFunctions.C`, that allows to use the function `flowrateModel`;
5. Create a `TLauncherFunction` object from the `TDataServer` and the analytic function `flowrateModel`. The inputs are all the `TAttribute` objects of the `TDataServer`, taken in the order of the calls to the `addAttribute` methods (r_ω, r, T_u, T_l, H_u, H_l, L, K_ω). **WARNING WITH ROOT6:** the second argument should be either a string or a pointer if the function is defined in the same file (see Section I.2.5) and the last argument that gives the name of the output attribute cannot be omitted anymore.
6. Define whether a progress bar should be drawn (here we switch it off).
7. Evaluate the function on the design-of-experiments contained in the `TDataServer` and store the output y in a new `TAttribute` whose name is the name of the function *flowrateModel* (we can see the use of the `flowrateModel` variable in the last `drawProfile` line).

At any moment, if one wants to verify the name of the function used, a call to the method `GetName` will return the name of the `TLauncherFunction` function:

```
cout << tlf->GetName() << endl; // will print "flowrateModel"
```

If the attributes have not been defined in the order they will be used in the function, the user must specify them in the right order in the third argument of the function as a string that contains the variables names separated by colons. In similar manner, if there are attributes that are defined but not useful for a launcher definition, in order Uranie not to take them into account, just specify the list of the useful attributes in the definition of the `TLauncherFunction`. The following example will produce the same result as the previous one:

```
// Create a TDataServer
TDataServer * tds = new TDataServer("tdsFlowrate","TDS for flowrate");
// Add the eight attributes of the study with uniform laws
tds->addAttribute( new TUniformDistribution("a", 0., 1.));          ❶[1]
tds->addAttribute( new TUniformDistribution("b", 0., 1.));
tds->addAttribute( new TUniformDistribution("c", 0., 1.));

tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));    ❷[2]
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

// Generate the sampling from the TDataServer
TSampling *sampling = new TSampling(tds, "lhs", 1000);
sampling->generateSample();

// Load the function in the UserFunction macros file
```

```
gROOT->LoadMacro("UserFunctions.C");

// Create a TLauncherFunction from a TDataServer and an analytic function
TString fName="flowrateModel";
TLauncherFunction *tlf = new TLauncherFunction(tds, fName, "rw:r:tu:tl:hu:hl:l:kw", fName); ↩
        ❸[3]

// Evaluate the function on all the design-of-experiments
tlf->setDrawProgressBar(kFALSE);
tlf->run();

tds->drawProfile("flowrateModel:rw");
```

Specification of the input variables for an analytic function

❶ Three useless attributes have been added to the `TDataServer`;

❷ The creation order of the other attributes have been mixed up;

❸ In order for Uranie to treat only the wanted attributes (not `a, b, c`), and in the right order, we add the string `"rw:r:tu:tl:hu:hl:l:kw"` in the definition of the `TLauncherFunction`.

Tip
Note that if the third argument is the empty string (`""`), the behaviour is the same as if the argument is not specified. That is useful to specify a fourth argument (see below) without having to specify the third.

By default, Uranie assumes the function returns only one value. In case the function returns several values and one wants to precise their name explicitely, it is necessary to specify their number to Uranie to create the different attributes in the `TDataServer`, by using the `setNOutput` method. For example if we use the following definition in the previous example:

```
tlf->setNOutput(2);
```

then, after running, two outputs would be available: `flowrateModel_1` and `flowrateModel_2`, whose names are created automatically from the function name and an index.

Finally, it is possible to specify the names of the output variables as the fourth argument of the `TLauncherFunction`. This is highly recommended, and mainly useful in two cases:

- for renaming the output variable in order to ease its manipulation (*e.g.* `y` instead of `flowrateModel`);
- for getting more than one output variable if the function returns several, without using the `setNOutput` method. As an example, assuming that the function returns two values (*i.e.* it defines `y[0]` and `y[1]`), we can have the following call, that will produce values for outputs named `y0`, `y1`:

```
TLauncherFunction *tlf2out = new TLauncherFunction(tds, "flowrateModel", "rw:r:tu:tl:hu:hl ↩
    :l:kw", "y0:y1");
```

As seen previously, the `TLauncherFunction::run` can be called without argument to run the function with the default inputs and outputs, but the method can also be called with arguments that specify the inputs and outputs to use for a specific computation. For example, assuming that we have added two sets of input attributes, [r^1_ω, r^1, T^1_u, T^1_l, H^1_u, H^1_l, L^1, K^1_ω] and [r^2_ω, r^2, T^2_u, T^2_l, H^2_u, H^2_l, L^2, K^2_ω], to the dataserver, then we could get the result of the computation on the first set, y1, and on the second set, y2, by using the following code:

```
tlf->run("rw1:r1:tu1:tl1:hu1:hl1:l1:kw1", "y1");
tlf->run("rw2:r2:tu2:tl2:hu2:hl2:l2:kw2", "y2");
```

More example of running functions with a *TLauncherFunction* object, are shown in the use-case chapter, from Section XIV.4.1 to Section XIV.4.3.

Summary: `TLauncherFunction` object

- `TLauncherFunction`(**`TDataServer`** *tds, **void*** function(**double** *,**double** *), **TString** sinput, **TString** soutput)

  Create a launcher with the analytic function `function` to apply on the `TDataServer tds` where the input attributes are specified in `sinput` and the outputs in `soutput`.

- `TLauncherFunction`(**`TDataServer`** *tds, **const char*** functionname, **TString** sinput="", **TString** soutput="")

  Create a launcher with the analytic function whose name is `functionname` (see Section I.2.5) to apply on the `TDataServer tds` where the input attributes can be specified in `sinput` and the outputs in `soutput`.

- `run`(**TString** sinput="", **TString** soutput="", **Option_t*** option="")

  Evaluate the function on the design-of-experiments with `sinput` as input attributes and `soutput` as output attributes.

  Warning
  These arguments are optional.

- `setNOutput`(**Int_t** n)

  The main interest of this method is to return the number of output variables in constructor (*soutput=""*), and, in this case, cost variables names will be concatenation between the name of the function and the index.

IV.3 External Code

In Uranie, an external code is defined by a command line, and input and output attributes which are stored in *ASCII* files, named respectively *input files* and *output files*.

If, because of the way the function are defined, a `TLauncherFunction` is bound to have only double-precision attributes as input, dealing with a code allows more flexibility and it is now possible to handle both vectors and strings as inputs and outputs.

To launch an external code, it is necessary to define a `TLauncher` object. This object is used to launch the code defined in a `TCode` object and that use `TInputFile` input files and `TOutputFile` output files (as shown in Figure IV.2).

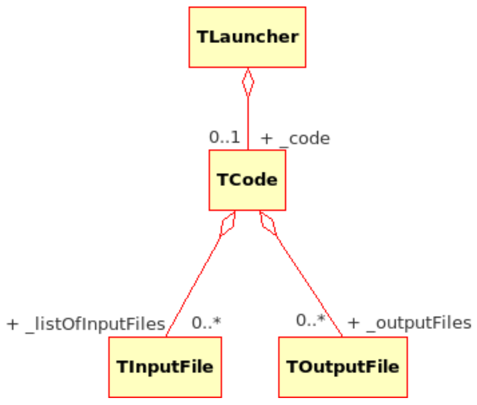


Figure IV.2: Inheritance diagram for the class TLauncher

This section is organised as follow: we first describe the definition of `TInputFile` and `TOutputFile` in Section IV.3.1, then the construction of a `TCode` in Section IV.3.2, and finally the construction of a `TLauncher` in Section IV.3.3. A schematic view of the code launching process is represented in Figure IV.3.

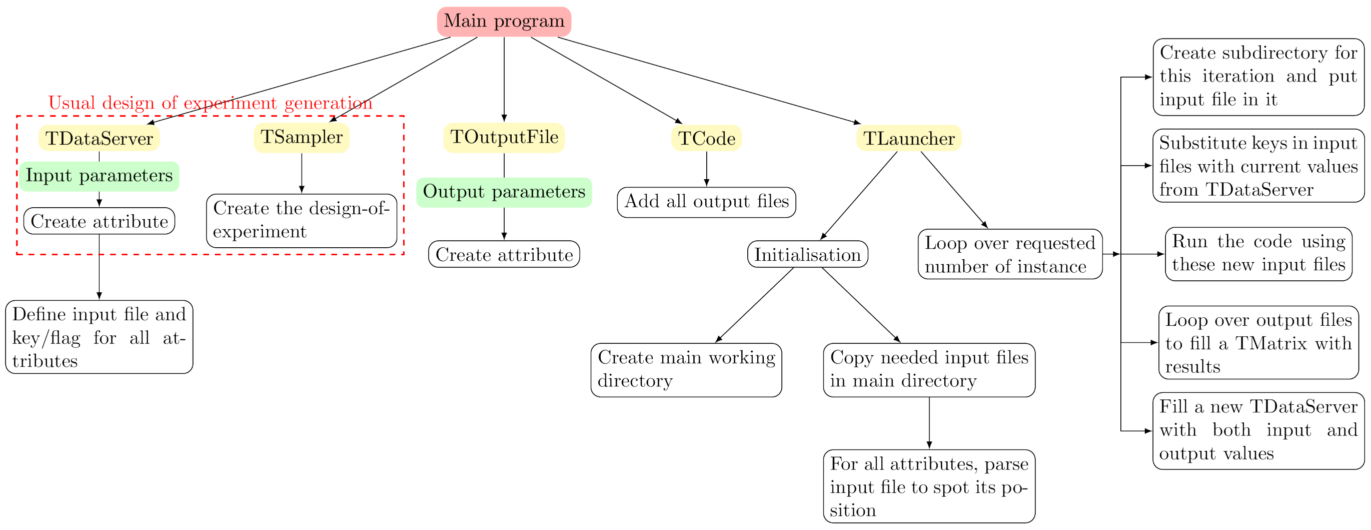

Main program
Usual design of experiment generation
TDataServer
Input parameters
Create attribute
TSampler
Create the design-of-experiment
Define input file and key/flag for all attributes
TOutputFile
Output parameters
Create attribute
TCode
Add all output files
TLauncher
Initialisation
Create main working directory
Copy needed input files in main directory
For all attributes, parse input file to spot its position
Loop over requested number of instance
Create subdirectory for this iteration and put input file in it
Substitute keys in input files with current values from TDataServer
Run the code using these new input files
Loop over output files to fill a TMatrix with results
Fill a new TDataServer with both input and output values

Figure IV.3: Schematic description of the launcher procedure when using an external code. Yellow boxes show instances of class, and green ones are precision about attributes. The design-of-experiments part can be replaced by an already-existing database.

The paragraphs below describe the main part of this procedure, as it is displayed by Figure IV.3, remaining at the class level. It can indeed be decomposed as follow:

- A design-of-experiments is produced (the corresponding part is highlighted in the red-dashed rectangle). On top of this, a file is "attached" to every input attribute, defining a way to spot the place where to find information when the launcher would have to run the code. This part is furthered discussed in Section IV.3.1.2
- One or more `TOutputFile` are created and the output attributes are attached to them, specifying, as for previous step, how to extract the information once the launcher will be running the code. This part is also described in more details in Section IV.3.1.3.

- A TCode is created by giving the input TDataServer along with the string used to launch the code in a terminal. The output files are then attached to the TCode object.
- The TLauncher object is created by giving the input TDataServer and the previously defined TCode. During the creation, a working directory is made where input files are copied. Then the loop is performed, running the code as many times as there are points in the design-of-experiments. This loop can be decomposed as well in few key-steps:
  1. An initialisation: a sub-directory is created for this specific computation, in which the input files are copied and modified to used the requested input values defined in the design-of-experiments and stored in the TDSNtupleD.
  2. Running the external code itself
  3. Parsing the output files to collect the desired values and send them back to the launcher which fills the ntuple of results. This ntuple is merged to the input one at the end of the process.

Precision on the described steps

- One can ask to use more than one process in order to proceed with the computation. In this case, the main launcher calls the TLauncherMulti to distribute events one-by-one. The results are gathered and the output TDataServer is saved every 5 events (disregarding the number of processes requested). This is further discussed in Section IV.4.
- The usual procedure is to write a file for every event processed (as most of the time the code to be run is slow) and back up the complete output file every 5 events (as said previously). In the unusual case where the code is fast, this step is considerably slowing down the process of running and collecting the outputs.

  It is possible to skip this step, by calling the run method passing the following optional argument: "noIntermediateSaved". This has to be used with caution as the output file will be opened only once at the very end of the process, so if this step is not reach for one reason or another, everything will be lost and one would have to re-run from the beginning.

IV.3.1 Code input and output files

Uranie can manage different kinds of input and output files. We'll describe below these different IO file formats after a short introduction on the way these files are implemented.

IV.3.1.1 General discussion

All the input/output files that will be discussed in the following sections (meaning restrained to the Launcher module) are based on a class named TamponTexte, which comprehends a file as a group of field, apart one to another thanks to separators. It is absolutely crucial to keep this is mind.

With the introduction of the vectors and strings in version 3.10.0, more complex interactions with files were introduced: how to differ two iterations of a single vector and how to differentiate a double from a string. This depends highly on the nature of the input/output file under consideration, whether it is just a text file used as database (in this case it depends mostly on the way you've written the code that generates/parses it) or whether it corresponds to a stricter kind of file, for instance a piece of code (c++/python/zsh). In the latter case, strings and vectors are not written in the same way. To take this into account, a rule has been defined (commonly to both input and output files, both in the Launcher and Relauncher module). There is a method for any kind of file to define properties of vector and string objects:

- `setVectorProperties(string beg, string delim, string end)`: the first element is the string beginning of the vector (usually "[" for python, "(" for zsh/sh, nothing...), the second one is the delimiter between iterators (usually "," for c++/python, blank for zsh/sh...) and the last one is the end of the string (usually the opposite character of the beginning one).
- `setStringProperties(string beg, string end)`: the first and second elements are respectively the beginning and ending character used for string (oftenly """).

In order to illustrate how to deal with a complicated case (when the order provided by the database is not the one needed by the code, but also dealing with vectors and strings as inputs), one can have a look at the examples provided from Section XIV.4.29 to Section XIV.4.33. The output files, on the other hand, are more thoroughly (and explicitly) discussed in Section IV.3.1.3.

IV.3.1.2 Input files

When a code needs input parameters from one or more input files, Uranie must be able to duplicate and modify these files in order to set the input parameter under discussion to its requested value. Uranie can do this using different kinds of input files. Before going through the following sections, which will present the way input files are used with the **flowrate** problem presented in Section IV.1.2, a small discussion will introduced the different kinds of input file, some of their properties and the way they are usually declared.

The classes used to either do the substitution in the input file or to recreate a new input file to feed the code, were not much discussed up to now in this documentation as they were created on the fly, by the `TCode` itself at the initialisation. This is done by the mean of checking the list of `EFileType` for every attribute (one attribute can indeed have more than one file and type in case the code, or codes, need to have this information in different files, with different formats). An explicit declaration is however possibly needed for two reasons:

- Letting the `TCode` object do, as described above, one can not choose the order to which the attribute are attached to the corresponding dumper object, and so, the order in which they will be written (which can be of uttermost importance for some format). Indeed the `TCode` object will follow the order in which the attribute have been added to the `TDataServer` by hand (but in the case the user just has to invert the corresponding attributes) or, more likely, through a database (*e.g.* using the `fileDataRead` method).
- In the case of vector and strings handling, it might be necessary to specify the delimiter, beginning and ending sign (as already introduced above and in the last part of Section II.3.1.1).

The classes to handles inputs files can be split into two different types:

- the one creating an input file from scratch: done with `TInputFileRecreate` class, it can use different implementations according to the type requested by the user, through the `TAttribute::EFileType` provided in the `setFileKey` method (discussed below). The possible values are: `kNewKey`, `kNewTDS`, `kNewRow` and `kNewColumn`. These flags are discussed below.
- the ones which modify an existing file: this is done with specific classes, depending on the type of inputs used by the code and this is also precised by the user through the `TAttribute::EFileType` provided in the `setFileKey` method. The possible values are: `kKey` (creating implicitly a `TInputFileKey` object), `kFlag` (creating a `TInputFileFlag` object), and both `kXMLAttribute` and `kXMLField` (used in the `TInputFileXML` instance).

IV.3.1.2.1 Input files with "key=value" format

Let us consider the **flowrate** input file with "key = value" format, with the eight parameters (r_ω, r, T_u, T_l, H_u, H_l, L, K_ω):

```
#
#
# INPUT FILE with KEYS for the "FLOWREATE" code
# \date   2008-04-22 12:53:35
#


date = 123456 ;


#########################
##
##  exclude points
##
chu = 1050;
chl = 770;
cr  = 1100;
##
#########################

#########################
##
##  parameters : 8
##
Rw = 0.0500 ;
R = 33366.67 ;
Tu = 63070.0 ;
Tl = 116.00 ;
Hu = 1110.00 ;
Hl = 768.57;
L = 1200.0 ;
Kw = 11732.14 ;
##
#########################


#########################
##
## to simulate CPU time
##
## normal          1 :
## min      10000000 : 1.160u 0.000s 0:01.16 100.0%
## max     100000000 : 11.600u 0.010s 0:11.61 100.0%
##
nLoop = 1;
##
#########################

end = 6;
```

Please note that this file has been created out of Uranie and that the values are not managed by Uranie yet: it is just a model of input file for the code.

Now we write the beginning of the Uranie script that is going to define attributes and assign them a place in the input file:

```
{
// Define the DataServer
```

```
TDataServer *tds = new TDataServer("tdsFlowrate", "Doe for Flowrate");

// Add the study attributes ( min, max and nominal values)
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));              ❶[1]
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

// The reference input file
TString sJDDReference = TString("flowrate_input_with_keys.in");               ❷[2]

// Set the reference input file and the key for each input attributes
tds->getAttribute("rw")->setFileKey(sJDDReference, "Rw");                     ❸[3]
tds->getAttribute("r")->setFileKey(sJDDReference, "R");
tds->getAttribute("tu")->setFileKey(sJDDReference, "Tu");
tds->getAttribute("tl")->setFileKey(sJDDReference, "Tl");
tds->getAttribute("hu")->setFileKey(sJDDReference, "Hu");
tds->getAttribute("hl")->setFileKey(sJDDReference, "Hl");
tds->getAttribute("l")->setFileKey(sJDDReference, "L");
tds->getAttribute("kw")->setFileKey(sJDDReference, "Kw");
...
```

Specification of a "key=value" input file for the input variables

❶ In this first part, definition of the attributes of the `TDataServer`

❷ Definition of the reference input file (the file copied before);

❸ At each of these lines, definition of the place the parameter will be set, thanks to the `setFileKey` method, that takes as parameters the input file name and the name of the variable to replace. For example, the "rw" parameter will be set in the file `flowrate_input_with_keys.in` where the `Rw` value stands.

After these definitions, if we create the code launcher and launch it (what will be presented in the following sections), then the input file would be modified by writing each sample value in a new input file. For example, the following input file is one of the generated ones:

```
#
#
# INPUT FILE with KEYS for the "FLOWREATE" code
# \date   2008-04-22 12:53:35
#


date = 123456 ;


#########################
##
##  exclude points
##
chu = 1050;
```

```
chl = 770;
cr  = 1100;
##
#########################

#########################
##
##  parameters : 8
##
Rw = 5.194485e-02 ;
R = 8.610663e+03 ;
Tu = 7.644908e+04 ;
Tl = 1.003881e+02 ;
Hu = 1.058603e+03 ;
Hl = 7.224698e+02;
L = 1.148747e+03 ;
Kw = 1.003128e+04 ;
##
#########################


#########################
##
## to simulate CPU time
##
## normal         1 :
## min     10000000 : 1.160u 0.000s 0:01.16 100.0%
## max    100000000 : 11.600u 0.010s 0:11.61 100.0%
##
nLoop = 1;
##
#########################

end = 6;
```

In this file, we will note the different values of the parameters, that have been modified compared to the "model" presented in the beginning of this section.

Tip
No matter the number of times the key appears in the input file, the value will only be replaced once!

IV.3.1.2.2 Input files with flag format

Let us consider the **flowrate** input file with flag format, with the eight parameters (r_ω, r, T_u, T_l, H_u, H_l, L, K_ω). As seen in Section IV.1.2.3.1.2, it is necessary to write "flags" where the values will be written, as in the example below:

```
#
# INPUT FILE with FLAG for the "FLOWREATE" code
# \date   2008-04-22 12:55:17
#

new Implicit_Steady_State sch {
    frottement_paroi { @Rw@ @R@ }
    tinit              0.0
    tmax               1000000.
    nb_pas_dt_max      1500
    dt_min              @Hu@
    dt_max              @Hl@
    facsec              1000000.
    kW                  @Kw@
    information_Tu      Champ_Uniforme 1         @Tu@
    information_Tl      Champ_Uniforme 1         @Tl@
    information_L {
        precision  @L@
    }
    convergence {
        criterion relative_max_du_dt
        precision  @Rw@
    }

    stop_criterium {
        ch_abcsissa_hu 1050
        ch_ordinate_hl 770
        c_radius 1100
    }

    Solveur Newton3 {
        max_iter_matrice        1
        max_iter_implicite      1
        date                    5654321
        seuil_convg_implicite   1.e-6
        assemblage_implicite    10
        solveur_lineaire        BiCGS
            preconditionneur    ILU
            seuil_resol_implicite 1.e-5
    }
}
```

Please note that this file has not been created out of Uranie since the flags have been written manually: this file is just a model of input files for the code.

Now we write the beginning of the Uranie script that is going to define attributes and to assign them a place in the input file:

```
// Define the DataServer
TDataServer *tds = new TDataServer("tdsFlowrate", "Doe for Flowrate");

// Add the study attributes ( min, max and nominal values)
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));       ❶[1]
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

// The reference input file
TString sJDDReference = TString("flowrate_input_with_flags.in");        ❷[2]

// Set the reference input file and the key for each input attributes
tds->getAttribute("rw")->setFileFlag(sJDDReference, "@Rw@");            ❸[3]
tds->getAttribute("r")->setFileFlag(sJDDReference, "@R@");
tds->getAttribute("tu")->setFileFlag(sJDDReference, "@Tu@");
tds->getAttribute("tl")->setFileFlag(sJDDReference, "@Tl@");
```

```
tds->getAttribute("hu")->setFileFlag(sJDDReference, "@Hu@");
tds->getAttribute("hl")->setFileFlag(sJDDReference, "@Hl@");
tds->getAttribute("l")->setFileFlag(sJDDReference, "@L@");
tds->getAttribute("kw")->setFileFlag(sJDDReference, "@Kw@");
```

Specification of a flag input file for the input variables

1. In this first part, definition of the attributes of the `TDataServer`;
2. Definition of the reference input file (the file copied before);
3. For every attribe, the position to write the information in the file is specified thanks to the `setFileFlag` method. It takes as first parameter the input file name and then the flag where the value will be set. For example, the "rw" parameter will be set in the file `flowrate_input_with_keys.in` where the `@Rw@` flag stands.

After these definitions, if we create the code launcher and run it (what will be presented in the following sections), then the input file would be modified by writing each sample value in a new input file. For example, the following input file is one of the generated ones:

```
#
# INPUT FILE with FLAG for the "FLOWREATE" code
# \date   2008-04-22 12:55:17
#

new Implicit_Steady_State sch {
    frottement_paroi { 6.371966e-02 4.913216e+04 }
    tinit               0.0
    tmax                1000000.
    nb_pas_dt_max       1500
    dt_min              1.086881e+03
    dt_max              7.372423e+02
    facsec              1000000.
    kW                  1.190644e+04
    information_Tu      Champ_Uniforme 1        7.628909e+04
    information_Tl      Champ_Uniforme 1        7.386808e+01
    information_L {
              precision  1.402143e+03
      }
    convergence {
              criterion relative_max_du_dt
                  precision  1.e-6
    }

    stop_criterium {
            ch_abcsissa_hu 1050
        ch_ordinate_hl 770
                  c_radius 1100
       }

    Solveur Newton3 {
                   max_iter_matrice        1
                   max_iter_implicite      1
           date                   5654321
                 seuil_convg_implicite   1.e-6
                   assemblage_implicite    10
                   solveur_lineaire        BiCGS
                   preconditionneur    ILU
```

```
                    seuil_resol_implicite 1.e-5
        }
}
```

In this file, we will note the different values of the parameters that have been modified compared to the "model" presented in the beginning of this section.

Tip
If a flag exists several times in a file, the value will be replaced at all the occurrences of the flag.

IV.3.1.2.3 Creation of input files without model files

In case the user does not want to create a model file, it is possible to fully create the input file. To do so, it is necessary to use the `setFileKey` method on an attribute and to specify the format of the file to be created. Uranie is able to generate input files with "key = value" format, "values in row" or "values in column".

- For the "key = value" format, the `setFileKey` method must be called with a string that describes the input format (see the C++ reference page for the available formats) as third parameter (if the string is empty, the default format will be used), and the flag `TAttributeFileKey::kNewKey` as fourth parameter. As an example the definition below:

  ```
  TString sIn = TString("test_input_with_keys.in");
  tds->getAttribute("rw")->setFileKey(sIn,"Rw","%e",TAttributeFileKey::kNewKey);
  tds->getAttribute("r")->setFileKey(sIn,"R","%e",TAttributeFileKey::kNewKey);
  tds->getAttribute("tu")->setFileKey(sIn,"Tu","%e",TAttributeFileKey::kNewKey);
  tds->getAttribute("tl")->setFileKey(sIn,"Tl","%e",TAttributeFileKey::kNewKey);
  tds->getAttribute("hu")->setFileKey(sIn,"Hu","",TAttributeFileKey::kNewKey);
  tds->getAttribute("hl")->setFileKey(sIn,"Hl","",TAttributeFileKey::kNewKey);
  tds->getAttribute("l")->setFileKey(sIn,"L","",TAttributeFileKey::kNewKey);
  tds->getAttribute("kw")->setFileKey(sIn,"Kw","",TAttributeFileKey::kNewKey);
  ```

  will lead in the creation of files `test_input_with_keys.in`, as:

  ```
  Rw = 1.136676e-01 ;
  R = 2.705506e+04 ;
  Tu = 9.259556e+04 ;
  Tl = 6.400877e+01 ;
  Hu = 1.087919e+03 ;
  Hl = 7.600170e+02 ;
  L = 1.537756e+03 ;
  Kw = 1.060694e+04 ;
  ```

- For the "value in row" format, the `setFileKey` method must be called with a string that describes the input format (see the C++ reference page for the available formats) as third parameter (if the string is empty, the default format will be used), and the flag `TAttributeFileKey::kNewRow` as fourth parameter. As an example the definition below:

  ```
  TString sInR = TString("test_input_with_row.in");
  tds->getAttribute("rw")->setFileKey(sInR,"Rw","%e",TAttributeFileKey::kNewRow);
  tds->getAttribute("r")->setFileKey(sInR,"R","%e",TAttributeFileKey::kNewRow);
  tds->getAttribute("tu")->setFileKey(sInR,"Tu","%e",TAttributeFileKey::kNewRow);
  tds->getAttribute("tl")->setFileKey(sInR,"Tl","%e",TAttributeFileKey::kNewRow);
  tds->getAttribute("hu")->setFileKey(sInR,"Hu","",TAttributeFileKey::kNewRow);
  ```

```
tds->getAttribute("hl")->setFileKey(sInR,"Hl","",TAttributeFileKey::kNewRow);
tds->getAttribute("l")->setFileKey(sInR,"L","",TAttributeFileKey::kNewRow);
tds->getAttribute("kw")->setFileKey(sInR,"Kw","",TAttributeFileKey::kNewRow);
```

will lead in the creation of files `test_input_with_row.in`, as:

```
1.136676e-01 2.705506e+04 9.259556e+04 6.400877e+01 1.087919e+03 7.600170e+02 1.537756e+03 ↩
     1.060694e+04
```

- Caution must be taken with vectors: before this version, the Row format and the DataServer format (for output files) were equivalent up to the headers. Now this is different, as vector are dumped on a single line for DataServer format (also called Salome-table) and with one value per line in Row files. A new flag as then be created as can be seen for instance when comparing Section XIV.4.30 and Section XIV.4.32. For the new DataServer format, the `setFileKey` method must be called with a string that describes the input format (see the C++ reference page for the available formats) as third parameter (if the string is empty, the default format will be used), and the flag `TAttributeFileKey::kNewTDS` as fourth parameter. As an example the definition below:

```
TString sInTds = TString("test_input_tds.in");
tds->getAttribute("rw")->setFileKey(sInTds,"Rw","%e",TAttributeFileKey::kNewTDS);
tds->getAttribute("r")->setFileKey(sInTds,"R","%e",TAttributeFileKey::kNewTDS);
tds->getAttribute("tu")->setFileKey(sInTds,"Tu","%e",TAttributeFileKey::kNewTDS);
tds->getAttribute("tl")->setFileKey(sInTds,"Tl","%e",TAttributeFileKey::kNewTDS);
tds->getAttribute("hu")->setFileKey(sInTds,"Hu","",TAttributeFileKey::kNewTDS);
tds->getAttribute("hl")->setFileKey(sInTds,"Hl","",TAttributeFileKey::kNewTDS);
tds->getAttribute("l")->setFileKey(sInTds,"L","",TAttributeFileKey::kNewTDS);
tds->getAttribute("kw")->setFileKey(sInTds,"Kw","",TAttributeFileKey::kNewTDS);
```

will lead in the creation of files `test_input_with_tds.in`, as:

```
#COLUMN_NAMES: rw|r|tu|tl|hu|hl|l|kw
#COLUMN_TYPES: D|D|D|D|D|D|D|D

1.136676e-01 2.705506e+04 9.259556e+04 6.400877e+01 1.087919e+03 7.600170e+02 1.537756e+03 ↩
     1.060694e+04
```

Once again, this file and the previous one looks very familiar, but the input file created with vectors, using respectively DataServer and Row format in Section XIV.4.30 and Section XIV.4.32 are not.

- For the "values in column" format, the `setFileKey` method must be called with a string that describes the input format (see the C++ reference page for the available formats) as third parameter (if the string is empty, the default format will be used), and the flag `TAttributeFileKey::kNewColumn` as fourth parameter. As an example the definition below:

```
TString sInC = TString("test_input_with_column.in");
tds->getAttribute("rw")->setFileKey(sInC,"Rw","%e",TAttributeFileKey::kNewColumn);
tds->getAttribute("r")->setFileKey(sInC,"R","%e",TAttributeFileKey::kNewColumn);
tds->getAttribute("tu")->setFileKey(sInC,"Tu","%e",TAttributeFileKey::kNewColumn);
tds->getAttribute("tl")->setFileKey(sInC,"Tl","%e",TAttributeFileKey::kNewColumn);
tds->getAttribute("hu")->setFileKey(sInC,"Hu","",TAttributeFileKey::kNewColumn);
tds->getAttribute("hl")->setFileKey(sInC,"Hl","",TAttributeFileKey::kNewColumn);
tds->getAttribute("l")->setFileKey(sInC,"L","",TAttributeFileKey::kNewColumn);
tds->getAttribute("kw")->setFileKey(sInC,"Kw","",TAttributeFileKey::kNewColumn);
```

will lead in the creation of files `test_input_with_column.in`, as:

```
1.136676e-01
2.705506e+04
9.259556e+04
6.400877e+01
```

```
1.087919e+03
7.600170e+02
1.537756e+03
1.060694e+04
```

IV.3.1.2.4 Input files with XML format

Uranie is also able to generate data in XML input files. For example, let us consider the following input file that contains the eight parameters previously seen (r_ω, r, T_u, T_l, H_u, H_l, L, K_ω):

```
<?xml version="1.0"?>
<problem>
<description name="flowrate" version="1.0" title="UseCase flowrate with XML input file"  ↩
    date="2008-04-22 12:55:17">
<tool name="uranie" version="0.3"/>
</description>

<steady_state name="sch">
<frottement_paroi rw="0.0500" r="33366.67"/>
<tinit>0.0</tinit>
<tmax>1000000</tmax>
<nb_pas_dt_max>1500</nb_pas_dt_max>
<parameter>
<tonode>mailleur</tonode><toport>dt_min</toport>
      <value><double>1110.00</double></value>
</parameter>
<parameter>
<tonode>mailleur</tonode><toport>dt_max</toport>
<value><double>768.57</double></value>
</parameter>
<facsec>1000000.</facsec>
<kW value="11732.14"/>
      <informations>
<parameter name="Tu">
<Champ_Uniforme>1</Champ_Uniforme>
<value><double>63070.0</double></value>
</parameter>
<parameter name="Tl">
<Champ_Uniforme>1</Champ_Uniforme>
<value><double>116.00</double></value>
</parameter>
<parameter name="L" precision="1200.0"/>
</informations>

<convergence>
<criterion>relative_max_du_dt</criterion>
<precision>1.e-6</precision>
</convergence>

<stop_criterium ch_abcsissa_hu="1050" ch_ordinate_hl="770" c_radius="1100" nLoop="1"/>

<solveur name="Newton3">
<max_iter_matrice>1</max_iter_matrice>
<max_iter_implicite>1</max_iter_implicite>
<date>5654321</date>
<seuil_convg_implicite>1.e-6</seuil_convg_implicite>
<assemblage_implicite>10</assemblage_implicite>
      <solveur_lineaire name="BiCGS">
<preconditionneur>ILU</preconditionneur>
<seuil_resol_implicite>1.e-5</seuil_resol_implicite>
```

```
</solveur_lineaire>
</solveur>

</steady_state>

</problem>
```

The transformation of an XML file is described by using XSLT transformation directives. When these directives have been described in the script, Uranie generates an XSLT transformation file, that will be directly applied on the input XML file to transform it. This XSLT file will describe how to copy the reference XML file to an XML file whose values are changed. So the description of the location of the values to change is naturally written with XSLT directives. Below is a short summary of the main XSLT commands:

Summary of XSLT instructions

- To specify an attribute, put a **"@"** before the attribute name (*e.g. @min*);
- Nothing is needed before a tag name (*e.g. parameter*), but to specify a tag tree, tags will be separated by **"/"** (*e.g. /value/double*);
- The expression of a condition is written between brackets **"[]"** (*e.g. [@name='E']*);
- To write composite conditions, use the **"and"** keyword between **"[]"** (*e.g. [tonode='mailleur' and toport='h_mm']*).

We can see above that there are two kinds of place a variable value may be changed: in an attribute or in a field. For Uranie to be able to modify the value, it is necessary to specify the kind of replacement to make, using the `TAttributeFileKey::kXMLAttribute` key for an attribute, or the `TAttributeFileKey::kXMLField` key for a field.

The example below demonstrates how to modify the attribute values in the file presented earlier:

```
TString sJDDReferenceXML = TString("flowrate_input_with_xml.xml");
tds->getAttribute("rw")->setFileKey(sJDDReferenceXML, "frottement_paroi/@rw", "%e",  ↩
    TAttributeFileKey::kXMLAttribute);
tds->getAttribute("r")->setFileKey(sJDDReferenceXML, "frottement_paroi/@r", "%e",  ↩
    TAttributeFileKey::kXMLAttribute);
tds->getAttribute("tu")->setFileKey(sJDDReferenceXML, "parameter[@name='Tu']/value/double", ↩
     "%e", TAttributeFileKey::kXMLField);
tds->getAttribute("tl")->setFileKey(sJDDReferenceXML, "parameter[@name='Tl']/value/double", ↩
     "%e", TAttributeFileKey::kXMLField);
tds->getAttribute("hu")->setFileKey(sJDDReferenceXML, "parameter[tonode='mailleur' and  ↩
    toport='dt_max']/value/double", "%e", TAttributeFileKey::kXMLField);
tds->getAttribute("hl")->setFileKey(sJDDReferenceXML, "parameter[tonode='mailleur' and  ↩
    toport='dt_min']/value/double", "%e", TAttributeFileKey::kXMLField);
tds->getAttribute("l")->setFileKey(sJDDReferenceXML, "parameter[name='L']/@precision", "%e" ↩
    , TAttributeFileKey::kXMLAttribute);
tds->getAttribute("kw")->setFileKey(sJDDReferenceXML, "kW/@value", "%e", TAttributeFileKey ↩
    ::kXMLAttribute);
```

Note that the key that describes the kind of replacement is different depending on whether the value is stored as an attribute (`kXMLAttribute`) or a field (`kXMLField`).

When Uranie is run, an XSLT file is created to describe the changes to make in the original XML file. Below is an example of such a generated XSLT file:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:template match="frottement_paroi/@rw">
<xsl:attribute name="rw">
<xsl:value-of select="0.133648"/>
</xsl:attribute>
</xsl:template>

<xsl:template match="frottement_paroi/@r">
<xsl:attribute name="r">
<xsl:value-of select="6989.96"/>
</xsl:attribute>
</xsl:template>

<xsl:template match="parameter[@name='Tu']/value/double">
<double>
<xsl:value-of select="63442"/>
</double>
</xsl:template>

<xsl:template match="parameter[@name='Tl']/value/double">
<double>
<xsl:value-of select="77.2926"/>
</double>
</xsl:template>

<xsl:template match="parameter[tonode='mailleur' and toport='dt_max']/value/double">
<double>
<xsl:value-of select="1051.48"/>
</double>
</xsl:template>

<xsl:template match="parameter[tonode='mailleur' and toport='dt_min']/value/double">
<double>
<xsl:value-of select="750.107"/>
</double>
</xsl:template>

<xsl:template match="parameter[name='L']/@precision">
<xsl:attribute name="precision">
    <xsl:value-of select="1543.31"/>
</xsl:attribute>
</xsl:template>

<xsl:template match="kW/@value">
<xsl:attribute name="value">
<xsl:value-of select="11745.4"/>
</xsl:attribute>
</xsl:template>

<!-- Copy all the rest of the file -->
<xsl:template match="node()|@*">
<xsl:copy>
<xsl:apply-templates select="node()|@*"/>
</xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

When applied to the `flowrate_input_with_xml.xml` reference input file, this XSLT file produces the XML file

Uranie will effectively use:

```
<?xml version="1.0"?>
<problem>
<description name="flowrate" version="1.0" title="UseCase flowrate with XML input file" ↩
    date="2008-04-22 12:55:17">
<tool name="uranie" version="0.3"/>
</description>

<steady_state name="sch">
<frottement_paroi rw="0.133648" r="6989.96"/>
<tinit>0.0</tinit>
<tmax>1000000</tmax>
<nb_pas_dt_max>1500</nb_pas_dt_max>
<parameter>
<tonode>mailleur</tonode><toport>dt_min</toport>
<value><double>750.107</double></value>
</parameter>
<parameter>
<tonode>mailleur</tonode><toport>dt_max</toport>
<value><double>1051.48</double></value>
</parameter>
<facsec>1000000.</facsec>
<kW value="11745.4"/>
<informations>
<parameter name="Tu">
<Champ_Uniforme>1</Champ_Uniforme>
<value><double>63442</double></value>
</parameter>
<parameter name="Tl">
<Champ_Uniforme>1</Champ_Uniforme>
<value><double>77.2926</double></value>
</parameter>
<parameter name="L" precision="1200.0"/>
</informations>

<convergence>
<criterion>relative_max_du_dt</criterion>
<precision>1.e-6</precision>
</convergence>

<stop_criterium ch_abcsissa_hu="1050" ch_ordinate_hl="770" c_radius="1100" nLoop="1"/>

<solveur name="Newton3">
<max_iter_matrice>1</max_iter_matrice>
<max_iter_implicite>1</max_iter_implicite>
<date>5654321</date>
      <seuil_convg_implicite>1.e-6</seuil_convg_implicite>
<assemblage_implicite>10</assemblage_implicite>
<solveur_lineaire name="BiCGS">
<preconditionneur>ILU</preconditionneur>
<seuil_resol_implicite>1.e-5</seuil_resol_implicite>
</solveur_lineaire>
</solveur>

</steady_state>

</problem>
```

Warning
Obviously, the use of the XML input needs an XSLT processor installed.

Tip
It is important to note that Uranie can manage inputs split between multiple input files, just by specifying the name of these different files, whatever the kind of input file under consideration.

Summary: Input files

- `setFileKey`(**TString** sfile, **TString** skey = "", **TString** sformatToSubstitute = "%e", **TAttributeFileKey::EFileType** sFileType = TAttributeFileKey::kKey)

  Defines:

  - The input file `sfile` where to change the values of the attribute.
  - The key `skey` to substitute in the input file, if different than the name of the attribute (which is the default value taken if an empty string is given). In case of a "flagged file" the key is the flag. In case of an XML file, the key is the XSLT path of the tag.
  - The format to substitute `sformatToSubstitute`; the default value `%e` means that scientific format is used.
  - The type of input files `sFileType`, whose default default value is `kKey`.

  Note that the behaviour depends on the `sFileType` flag: the input file must exist for `kKey`, `kFlag`, `kXMLAttribute` or `kXMLField`, or will be created for `kNewRow`, `kNewColumn`, `kNewKey` or `kNewTDS`.

- `setFileFlag`(**TString** sfile, **TString** sflag = "", **TString** sformatToSubstitute = "%e")

  Equivalent to `setFileKey(sfile, skey, sformatToSubstitute, TAttributeFileKey::kFlag),` defines an input file with flags `sflag`.

IV.3.1.3 Output files

After the code has run, it has created output files we want to extract values from. We present here the different kinds of output files Uranie can manage, and how to retrieve information from them.

Uranie can retrieve information from files whose formats are "key = value", "values in row", "values in column", "dataserver value" and XML.

IV.3.1.3.1 Output files with "key=value" format

`TOutputFileKey` retrieves data from a file where values are stored in the format "key = value". An example of such a file is given below:

```
yhat = 6.757218e+01;
d = 4.092561e+03;
```

To get data from such a file, it is necessary to create a `TOutputFileKey` object, and add to it the attributes of the file:

```
TOutputFileKey *foutk = new TOutputFileKey("_output_flowrate_withKey_.dat");
foutk->addAttribute(new TAttribute("yhat"));
foutk->addAttribute(new TAttribute("d"));
```

When working with "key = value" files, the default name of the attribute to create (here `yhat` or `d`) is the name of the attribute in the output file. In order to create an attribute with a name different from the name in the file, it is necessary to create an attribute and set it to the output file:

```
TAttribute *attr = new TAttribute("y");
attr->setFileKey("_output_flowrate_withKey_.dat", "yhat");
foutk->addAttribute(attr);
```

It is also possible to retrieve only the values of the desired variables. In the previous example, it would have been possible to retrieve only the values of `d` just by deleting the line that treats `yhat` output.

IV.3.1.3.2 Output files with row format

`TOutputFileRow` retrieves data from a file where values are stored in rows. An example of such a file is given below:

```
#COLUMN_NAMES: yhat | d

6.757218e+01  4.092561e+03
```

To get data from such a file, it is necessary to create a `TOutputFileRow` object, and add to it the attributes of the file:

```
TOutputFileRow *foutr = new TOutputFileRow("_output_flowrate_withRow_.dat");
foutr->addAttribute(new TAttribute("yhat"));
foutr->addAttribute(new TAttribute("d"));
```

When working with "values in row" files, the addition order of the attributes determines which value will be stored in each variable. In the previous example, the first value encountered in the file will be stored in the `yhat` attribute, and the second one in `d`. As a consequence, in order to get only the values from a given column, it is necessary to specify the column number (first column is number 1, not 0). When this number is not specified, the value used is the one following the previous value. For example, if we suppose the file `test.dat` contains 8 values stored in row, we can have:

```
foutr = new TOutputFileRow("test.dat");
foutr->addAttribute(new TAttribute("y2"), 2);  // get the 2nd column value
foutr->addAttribute(new TAttribute("y3"));     // get the 3rd column value
foutr->addAttribute(new TAttribute("y8"), 8);  // get the 8th column value
foutr->addAttribute(new TAttribute("y5"), 5);  // get the 5th column value
foutr->addAttribute(new TAttribute("y6"));     // get the 6th column value
```

Warning If one is not considering vectors, Row and DataServer (Salome-table) format output are equivalent. Else, it differs a lot as it can be seen by comparing Section XIV.4.25 and Section XIV.4.27.

IV.3.1.3.3 Output files with column format

`TOutputFileColumn` retrieves data from a file where values are stored in columns. An example of such a file is given below:

```
2.618019e+01
3.602045e+03
```

To get data from such a file, it is necessary to create a `TOutputFileColumns` object, and add to it the attributes of the file:

```
TOutputFileColumn *foutc = new TOutputFileColumn("_output_flowrate_withColumn_.dat");
foutc->addAttribute(new TAttribute("yhat"));
foutc->addAttribute(new TAttribute("d"));
```

When working with "values in column" files, the addition order of the attributes determines which value will be stored in each variable. In the example, the first value encountered in the file will be stored in the `yhat` attribute, and the second one in `d`. As a consequence, in order to get only the values from a given row, it is necessary to specify the row number (first row is number 1, not 0). When this number is not specified, the value caught is supposed to have the next index that the previous value. For example, if we suppose the file `test.dat` contains 8 values stored in column, we can have:

```
foutc = new TOutputFileColumn("test.dat");
foutc->addAttribute(new TAttribute("y2"), 2);  // get the 2nd row value
foutc->addAttribute(new TAttribute("y3"));     // get the 3rd row value
foutc->addAttribute(new TAttribute("y8"), 8);  // get the 8th row value
foutc->addAttribute(new TAttribute("y5"), 5);  // get the 5th row value
foutc->addAttribute(new TAttribute("y6"));     // get the 6th row value
```

IV.3.1.3.4 Output files with DataServer format

Another output format is the format of the `TDataServer`: the first line describes the data and the data follows.

```
#COLUMN_NAMES: yhat | d

6.757218e+01  4.092561e+03
```

To get data from such a file, it is necessary to create a `TOutputFileDataServer` object, and add to it the attributes to be read from the file:

```
TOutputFileDataServer *foutds = new TOutputFileDataServer("_output_flowrate_withRow_.dat");
foutds->addAttribute(new TAttribute("yhat"));
foutds->addAttribute(new TAttribute("d"));
```

Here the order in which the parameters are retrieved is not important: when looking for the `"d"` attribute for example, as the name "d" is the second name of the `#COLUMN_NAMES:` line, Uranie will retrieve the second value of the line (*i.e.* `3.602045e+03`).

Warning If one is not considering vectors, Row and DataServer (Salome-table) format output are equivalent. Else, it differs a lot as can be seen by comparing Section XIV.4.25 and Section XIV.4.27.

IV.3.1.3.5 Output files with XML format

The last output format is the XML format, where values can be retrieved from XML fields or attributes. Note this kind of output needs the LibXML2 library.

```xml
<?xml version="1.0"?>
<steady_state name="flowrate">
  <parameter>
    <tonode>mesher</tonode>
    <toport>dt_hl</toport>
    <value>
      <double>2.618019e+01</double>
    </value>
  </parameter>
  <distance value="3.602045e+03"/>
</steady_state>
```

To get data from such a file, it is necessary to create a `TOutputFileXML` object, and add to it the attributes to catch from the file. When adding attributes, it is necessary to pass the name of the attribute to create or the attribute itself to the `addAttribute` method, as well as its XSL path and the kind of XML object to get data between `kXMLAttribute` and `kXMLField`. For example to get values from the XML file generated by `flowrate`, use the following code:

```cpp
TOutputFileXML *foutXML = new TOutputFileXML("_output_flowrate_withXML_.dat");
foutXML->addAttribute(TString("yhat"), "/steady_state[@name='flowrate']/parameter/value/ ↩
    double", URANIE::DataServer::TAttributeFileKey::kXMLField);
foutXML->addAttribute(new URANIE::DataServer::TAttribute("d"), "/steady_state[@name=' ↩
    flowrate']/distance/@value", URANIE::DataServer::TAttributeFileKey::kXMLAttribute);
```

As for the "key=value" and the DataServer formats, the order the parameters are provided is not important.

Summary: Output files

- `TOutputFileKey`(**TString** str)

  Create an object to read information from an ASCII file with the "key = value" format.

  `addAttribute`(**TAttribute** * att, **TString** skey="")

  Add the attribute `att` to the file and specify its key `skey`.

- `TOutputFileRow`(**TString** str)

  Create an object to read information from an ASCII file with the row format.

  `addAttribute`(**TAttribute** * att, **Int_t** nrow=0)

  Add the attribute `att` to the file and specify its row number `nrow`.

- `TOutputFileColumn`(**TString** str)

  Create an object to read information from an ASCII file with the column format.

  `addAttribute`(**TAttribute** * att, **Int_t** nline=0)

  Add the attribute `att` to the file and specify its line number `nline`.

- `TOutputFileDataServer`(**TString** str)

  Create an object to read information from an ASCII file with the DataServer formalism.

  `addAttribute`(**TAttribute** * att, **Int_t** nline=0)

  Add the attribute `att` to the file and specify its line number `nline`.

- `TOutputFileXML`(**TString** str)

  Create an object to read information from an XML file.

  `addAttribute`(**TAttribute** * att, **TString** xmlPath, **TAttributeFileKey::EFileType** FileType)

  Add the attribute `att` to the file, specify the XML path of the data `xmlPath`, and the type of XML data `FileType`, which can be `kXMLAttribute` or `kXMLField`.

IV.3.2 TCode definition

A `TCode` object is used to define an external code. In order to create such an object, the user must call the constructor whose arguments are a pointer on the useful `TDataServer` and a `TString` that contains the command to execute the external code. For example:

```
TCode *mycode = new TCode(tds, "flowrate -s -k");
```

This line will create the `TCode` object `mycode`, link it with the `TDataServer`, and assign the command line `flowrate -s -k` to it.

Some parameters can be set to the code: its working directory (Section IV.3.2.1), the definition of unmodified input files (Section IV.3.2.2), and the location of output files (Section IV.3.2.3).

IV.3.2.1 Working directory of the computations

By default, the computations are made in a new directory named `URANIE` in the current folder. To modify this behaviour, it is possible to specify the working directory by using the `setWorkingDirectory` method:

```
mycode->setWorkingDirectory(gSystem->pwd()+TString("/working_directory"));
```

With the previous line, the code will run in sub-directories of the newly created `working_directory` folder.

Tip
It is also possible to specify the working directory by using the `setWorkingDirectory` method on a `TLauncher` object (see Section IV.3.3). In case both methods are used, the code will use the path defined in the `TLauncher`, then the one defined in the `TCode`.

IV.3.2.2 Unmodified input files

While the variables with modifiable values are stored in the `TDataServer` (see Section IV.3.1), the code may need input files that contain parameters Uranie does not have to modify. To add such a file to the `TCode` object, just use the `addInputFile` method after having specified the directory to find it with the `setReferenceDirectory` method.

```
TCode *mycode2 = new TCode(tds, "flowrate -s -k");
mycode2->setReferenceDirectory( gSystem->pwd()+TString("/reference_directory") );
mycode2->addInputFile("flowrate_input_with_flags_1.in");
mycode2->addInputFile("flowrate_input_with_flags_2.in");
```

The previous lines will lead in the copy of the files `$PWD/reference_directory/flowrate_input_with_flags` and `$PWD/reference_directory/flowrate_input_with_flags_2.in` in the working directory of the code. Note that the reference directory definition should be made with an absolute path.

The method `getReferenceDirectory` will return this reference directory as a `TString`.

IV.3.2.3 Output file of the code

In order to specify the name of the output file of the code, it is necessary to use the `addOutputFile(outfile)` method, that defines the `TOutputfile` object `outfile` as an output file for the code:

```
foutk = new TOutputFileKey("_output_flowrate_withKey_.dat");
TCode *mycode3 = new TCode(tds, "flowrate -s -k");
mycode3->addOutputFile( foutk );
```

After having created the output file object `fout`, this code adds it as an output file to the `mycode` `TCode` object.

Summary: `TCode` object

- `TCode`(**TDataServer** * tds, **TString** scmd)

  Create a `TCode` object from the `TDataServer` `tds` and with the command line `scmd`.

- `setWorkingDirectory`(**TString** str)

  Set the directory for the code to run in. If not used, the code runs in the `URANIE` folder created in the current folder.

- `addInputFile`(**TString** str)

  Add the input file `str` to the `TCode` object.

- `addOutputFile`(**TOutputFile** * ifile)

  Add the (previously created) output file `ifile`. `ifile` can be an object of the following classes: `TOutputFileRow`, `TTOutputFileColumn`, `TTOutputFileKey`, `TTOutputFileDataServer` and `TOutputFileXML`

IV.3.3 Launcher definition

In order to run an external code, it is necessary to define an object that can launch this code. This object is a `TLauncher`. To create it, the constructor just needs a `TDataServer` and a `TCode` reference, as shown in the example below:

```
TDataServer *tds = new TDataServer("tdsFlowrate", "Doe for Flowrate");   ❶[1]
// (add data to the TDataServer)

TCode *mycode = new TCode(tds, "command");                               ❷[2]

TLauncher *mylauncher = new TLauncher(tds, mycode);                      ❸[3]
```

Creating a `TLauncher`

❶ Creation of a `TDataServer`. Then attributes and data can be appended (for further explanations and examples, please refer to the Chapter II).

❷ Creation of a `TCode`. See the Section IV.3.2 for a presentation of the construction of a `TCode` object.

❸ Construction of a `TLauncher`. As presented before, the constructor only needs the pointers on the `TDataServer` and on the `TCode`.

In order to manipulate the code, different methods can be used:

- `setWorkingDirectory`: set the directory in which the code will be run and where the temporary folders needed by the code will be stored.

  ```
  mylauncher->setWorkingDirectory( gSystem->Getenv("HOME") + TString("/tmp/"));
  ```

  The code runs in `$HOME/tmp/`. Note that, as seen in Section IV.3.2.1, if this function is not called on the current `TLauncher` object, the working directory can be set on the `TCode` object. If none of these methods are used, it will be set to a directory named `URANIE` created in the reference directory.

  The method `getWorkingDirectory` will return this directory as a `TString`.

- `setSave(i)`: activate the save mode and set the maximum number *i* of folders to be kept. A positive value for *i* stands for the explicit maximum number of saves, while no value or -1 value allow to keep all the patterns (all patterns are launched in a different working directory `UranieLauncher_i`). Default parameter is -1.

  If `setSave` is not called, the temporary output folders of the code will be overwritten at each computing step, while they will be left unmodified if the method is called.

```
mylauncher->setSave();     // will save each launch
mylauncher->setSave(-1);   // will save each launch
mylauncher->setSave(10);   // will save 10 launches
```

  The method `getSave` will return a boolean describing the state of the save flag, and the method `unsetSave` will deactivate the save mode.

Warning
In case of multi-process distributed computation, in order to prevent all the processes from trying and accessing the same data at the same time, a call to the `setSave` method is mandatory. As a consequence, if it is not manually set, Uranie will set it automatically.

- `setClean(kTrue/kFalse)`: set the clean flag to the value true or false. If it is set to `kTrue`, all the working directories are cleaned before launching. Then if the save flag is set to `kTRUE`, the working directories are cleaned a second time after the `TLauncher` run. If not specified, default parameter is `kTrue`, which means the working directory will be cleaned (*i.e.* temporary folders will be erased).

```
mylauncher->setClean();       // set the clean flag to "true"
mylauncher->setClean(kTRUE);  // set the clean flag to "true"
mylauncher->setClean(kFALSE); // set the clean flag to "false"
```

  The method `getClean` will return the clean flag as a boolean.

- `setVarDraw(str,select,opt)`: activate and define the graphics during the launching. By default, there is no graphic during the launching process. To activate it, it is necessary to define the information of the graphics (the X, Y and Z attributes, the selection and the options of the graphic). The method's parameters are:

  – `str`: the list of attributes to be drawn;

  – `select`: the selection of patterns to visualise (default value is "");

  – `opt`: the option of the graphics. Default value is "". To see all available values, please refer to the ROOT THistPainter reference page.

```
mylauncher->setVarDraw("hu:hl");                  // draw hu function of hl
mylauncher->setVarDraw("hu:hl","yhat>0");         // same for data where yhat>0
mylauncher->setVarDraw("hu:hl","yhat>0","SURF"); // same drawn as surface plot
```

- `setDrawProgressBar(bool)`: activate and draw the progress bar that gives an idea of the remaining time before finishing the loop. The default is to have this bar drawn, so this method is usually only used when one decides not to displays the progress bar (if the output is redirected in an output file for instance).

```
mylauncher->setDrawProgressBar();             // draw the progress bar
mylauncher->setDrawProgressBar(false);        // do not draw the progress bar
bool draw = mylauncher->getDrawProgressBar(); // check if progress bar is drawn
```

- After all the Launcher elements are set, it is then possible to launch it, by using the `run` method.

```
mylauncher->run();
```

We present below an example that uses the different methods described above:

```
TDataServer *tds = new TDataServer("tdsFlowrate", "DataBase flowrate");
// (add data to the TDataServer)

TCode *mycode = new TCode(tds, "command");

TLauncher *mylauncher = new TLauncher(tds, mycode);

mylauncher->setSave();
mylauncher->setClean();
mylauncher->setVarDraw("hu:hl","yhat>0","");
mylauncher->setWorkingDirectory(gSystem->Getenv("PWD") + TString("/tmpLanceurUranie/ ↩
    flowrate"));

mylauncher->run();
```

Summary: `TLauncher` object

- `TLauncher(`**`TDataServer`** *tds, **`TCode`** *tc)

  Create a launcher with the external code `tc` to apply on the `TDataServer` `tds`.

- `setWorkingDirectory(`**TString** str)

  Sets the working directory where to realise the computations to `str`.

- `setSave(`**Int_t** nSave)

  Activates the save mode and sets number of max saves to perform: a positive value stands for the explicit value of max saves, -1 allows to set max number of saves to the number of patterns.

- `setClean(`**Bool_t** bbool=kTRUE)

  Set the clean flag, which means that before launching, all the working directory are cleaned, and, if save is FALSE, after the launching all the working directory are also cleaned.

- `setVarDraw(`**TString** str, **TString** select="", **TString** opt="")

  Activates and defines the graphics during the launching. By default, there is no graphic during the launching process. To activate it, you must defines the information of the graphics: the X, Y and Z attributes, the select and the options of the graphic.

- `setDrawProgressBar(`**`Bool_t`** `bbool=kTRUE)`

  Activates and draws the progress bar that gives an idea of the remaining time before finishing the loop. The default is to have this bar drawn.

- `run()`

  Runs the code.

IV.4 Distribution

To reduce elapsed time of computation, as the computations are independent, we can execute in parallel several of these computations; it is the computing **distribution** scheme.

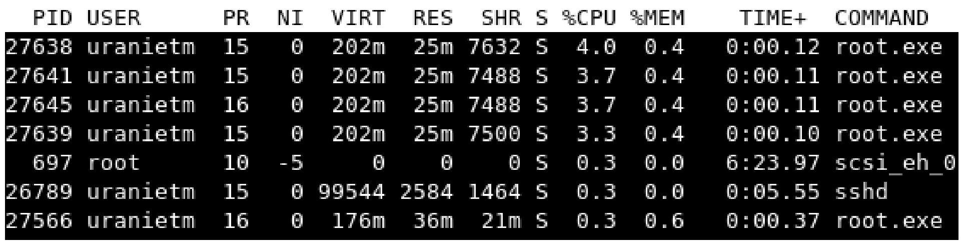
IV.4.1 Multi-core computer

In order to launch an Uranie script on a multi-core computer, the only modification to the code is the addition of the parameter `"localhost=X"` to the `run` function of the Launcher object (where `X` stands for the number of processors to use).

For example, the macro `launchCodeFlowrateKeySampling.C` can be launch on 5 processes with the following `run` definition:

```
mylauncher->run("localhost=5");
```

We can verify 5 processes are launched:



```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
27638 uranietm  15   0  202m  25m 7632 S  4.0  0.4   0:00.12 root.exe
27641 uranietm  15   0  202m  25m 7488 S  3.7  0.4   0:00.11 root.exe
27645 uranietm  16   0  202m  25m 7488 S  3.7  0.4   0:00.11 root.exe
27639 uranietm  15   0  202m  25m 7500 S  3.3  0.4   0:00.10 root.exe
  697 root      10  -5     0    0    0 S  0.3  0.0   6:23.97 scsi_eh_0
26789 uranietm  15   0 99544 2584 1464 S  0.3  0.0   0:05.55 sshd
27566 uranietm  16   0  176m  36m  21m S  0.3  0.6   0:00.37 root.exe
```

Figure IV.4: Multi-core computer

IV.4.2 Cluster

Uranie can also be launched on clusters with SLURM (curie at CCRT), LSF (tantale, platine at CCRT) or SGE (mars) with *BSUB* directives. Moreover, the same macro of Uranie is used when launched in a distributed mode or in a serial mode. It is only necessary to create a specific *job* file. It is also possible to use batch clusters in order to run several instances of a parallel code, mixing two levels of parallelism: one for the code, one for the design-of-experiments. This is achieved by specifying the number of cores to be used per job with the `setProcsPerJob(nbprocs)` method of `TLauncher`.

```
TCode *mycodeSlurm = new TCode(tds, "flowrate -s -f");
TOutputFileKey *foutSlurm = new TOutputFileKey("_output_flowrate_withKey_.dat");
mycodeSlurm->addOutputFile( foutSlurm );
TLauncher *mylauncherSlurm = new TLauncher(tds, mycodeSlurm);
mylauncherSlurm->setSave();
mylauncherSlurm->setClean();
mylauncherSlurm->setProcsPerJob(4);
mylauncherSlurm->setDrawProgressBar(kFALSE);
mylauncherSlurm->run();
```

This Uranie script excerpt will result in the execution of jobs with the command where `thecommand` is the command to be typed to run the code under study (either a `root -l -q script.C` or an executable if the code under consideration has been compiled).

IV.4.2.1 LSF clusters

An example of a *job* file is given below:

```
#BSUB -n  10                                                    ❶[1]
#BSUB -J  FlowrateSampling                                      ❷[2]
#BSUB -o  FlowrateSampling.out                                  ❸[3]
#BSUB -e  FlowrateSampling.err                                  ❹[4]

# Environement variables                                        ❺[5]
source uranie-platine.cshrc

# Clean the output file of bsub
rm -rf FlowrateSampling.out

# Launch the 1000 points of the design-of-experiments in 10 proc
root -l -q launchCodeFlowrateSampling.C
```

Example of LSF cluster run

❶ Define the number of processes to use, here 10.

❷ Define the name of the job, *FlowrateSampling*

❸ Name of the output file for the job.

Warning
If the `#BSUB -o` line is forgotten, the output will be sent by email.

❹ Name of the error output file for the job.

❺ All the lines following the *BSUB* instructions define the commands each node must run.

Once the job file is created, it can be launch by using the command:

```
bsub < BSUB_File
```

where `BSUB_FILE` is the name of the file created below.

In order to see one's own jobs, run:

```
bjobs
```

IV.4.2.2 SGE clusters

An example of a *job* file is given below:

```
#$ -S /bin/csh                                           ❶[1]
#$ -cwd                                                  ❷[2]
#$ -q express_par                                        ❸[3]
#$ -N testFlowrate                                       ❹[4]
#$ -l h_rt=00:55:00                                      ❺[5]
#$ -pe openmpi 16                                        ❻[6]

######################################################
###### Cleaning
rm -f FlowrateFGA.*.log _flowrate_sampler_launcher_.* *~  ❼[7]
rm -fr URANIE

######################################################
###### Execution
root -l -q launchCode.C
###### End Of File
```

Example of SGE cluster run

❶ Define the shell to use (here `CSH`).

❷ Run the job from the current working directory. Allow to keep the environment variables defined.

❸ Specify the queue to use (here `express_par`).

❹ Name the job (here `testFlowrate`).

❺ Maximum time the job will last, in hh:mm:ss format.

❻ All the lines following the *QSUB* instructions define the commands each node must run.

❼ All the lines following the *QSUB* instructions define the commands each node must run.

Once the job file is created, it can be launched by using the command:

```
qsub QSUB_FILE
```

where `QSUB_FILE` is the name of the previously-created file.

To see the running and pending jobs of the cluster, run the command:

```
qstatus -a
```

In order to see only one's own jobs:

```
qstat
```

IV.4.3 Advanced usage of batch systems

The execution of large number of runs on a batch machine can sometimes require adjustments. The mechanism employed by Uranie relies on low-level mechanisms which are piloted from the ROOT process. This can lead to bottlenecks or performance degradation. Also, the execution of many processes at the same time can put a heavy burden on the file system. The following precautions should therefore be taken:

- Standard output of the different processes should be kept at a reasonable level.
- IO should be made as much as possible on the local disks.
- When large number of processes run, the memory of the master node which runs jobs and also manages the execution can saturate. Uranie gives a possibility to dedicate this master node entirely to the execution management: `launcher->setEmptyMasterNode();`
- When large number of processes (more than 500 for instance) run, they can terminate simultaneously and consequently the system has difficulties detecting the end of the jobs. It is useful to use a temporising mechanism provided by the `setDelay(nsec)` method. This will make the last job start `nsec` seconds after the first one.

IV.4.4 Multi-step launching mechanism

In order to distribute computation the `TLauncher run` method creates directories, copies files, executes jobs and creates the output DataServer. All these operations are performed simultaneously, so that it is possible to delete execution directories as the computation are performed (see the use of the `setSave(Int_t nb_save)`).

However, it can be interesting to separate these operations when some of the runs fail or for batch systems. The `TLauncherByStep`, inherited from `TLauncher` does just that: instead of the `run` method, it has three methods which must be called sequentially:

- `preTreatment()` which creates the directories and prepares the input files before execution,
- `run(Option_t* option)` which performs the execution of the code,
- `postTreatment()` which retrieves the information from the output files and fills the `TDataserver`.

The `run` method can be called with the following options:

- option `"curie"` will use the SLURM exclusive mechanism to perform the computation on the curie TGCC machine. In this case, unlike the `TLauncher` mechanism, the root script should be called directly on the interactive node, and the script will create the batch file and submit it.

```
TCode *mycode = new TCode(tds, "flowrate -s -f");
TOutputFileKey *fout = new TOutputFileKey("_output_flowrate_withKey_.dat");
mycode->addOutputFile( fout );
TLauncherByStep *mylauncherBS = new TLauncherByStep(tds, mycode);
mylauncherBS->setSave();
mylauncherBS->setClean();
mylauncherBS->setProcsPerJob(4);
mylauncherBS->preTreatment();
mylauncherBS->setDrawProgressBar(kFALSE);
mylauncherBS->run("curie");
```

  After the batch is completed, the assembly of the DataServer can be achieved by calling the `postTreatment` method.

```
mylauncherBS->postTreatment();
```

IV.4.5 Multi-step remote launching to clusters

This is a new way to distribute computation on one or several clusters. The idea is very specific to some specific running conditions, summarised below:

- the cluster(s) must be reachable through ssh connections: Uranie has to be compiled, on the local machine you're working on, with a `libssh` library (whose version must be greater than 0.8.1).
- the code to be run has to be installed on the remote cluster(s) (with the same version, but this is up to the user to be sure of it).
- the cluster(s) on which one wants to run, must be SLURM-based (so far that is the only solution implemented).
- the job submission strategy of the cluster(s) have to allow the user to submit many jobs. The idea is indeed to run the estimations of the design-of-experiments by splitting in many jobs (up to one per locations) and send these jobs one by one through SSH tunneling in a given queue on the given cluster.

The main interesting consequence of this is that it allows to use clusters on which Uranie has not been installed on, as long as the user has an account and credential on it, an his code is accessible there as well.

Apart from the way the distribution is done, which is very specific and discussed below, it internal logic follows the example provided in Section IV.4.4 as it can be used to split the operations when some of the runs fail or for batch systems. The `TLauncherByStepRemote` indeed inherits from `TLauncher` and it contains three methods which must be called sequentially:

- `preTreatment()` which creates the directories and prepares the input files before execution,
- `run(Option_t* option)` which performs the execution of the code,
- `postTreatment()` which retrieves the information from the output files and fills the `TDataserver`.

The new steps are now discussed in the following subparts.

IV.4.5.1 Generate a header for scheduler

This generates and sends the scheduler a header file, produced thanks to a skeleton that is filled with information provided within the code and can be used by single and remote job submission. This skeleton will only differ depending on the scheduler used by the HPC platform that the user wants to use. It contains a set of common options that can be replaced within the macro file. The following file is an example

```
#!/bin/bash
#################################################################
##################MARENOSTRUM 4 BASIC HEADER###################
#################################################################

#######################SLURM DIRECTIVES#########################
# @filename@
#SBATCH -J @filename@
#SBATCH --qos=@queue@
#SBATCH -A @project@
#SBATCH -o @filename@.%j.out
#SBATCH -e @filename@.%j.err
#SBATCH -t @wallclock@
#SBATCH -n @numProcs@

###################END SLURM DIRECTIVES#####################
source @configEnv@
```

Additionally, the user can edit that file to add additional options always following the variable nomenclature `@directive_name@`. The skeleton will be handled internally by Uranie using the cluster configuration defined by the user within the macro.

IV.4.5.2 Define the propertie of the launcher(s)

The next steps is to configure the launcher or launchers (as one can split the bunch of computations to be done by creating several instances of `TLauncherByStepRemote`). In order to do this, a number of function has been implemented such as:

- tlch->setNumberOfChunks(Int_t numberofChunks): define the number of chunk;
- tlch->setJobName(TString path): define the job name
- The following lines are defining the compiling option as launchers can send code and compile it locally (new features for this remote launching): these following method will fill the `CMakeLists.txt.in` that is shown below.
  - tlch->addCompilerDirective(TString directive, TString value): tlch->addCodeExternalLibrary(TString libName, TString libPath="" , TString includePath=""); tlch->addCodeLibrary(TString libName, TString sourcesList); tlch->addCodeDependency(T destinationLib, TString sourceLibList);
- tlch->run("nsplit=<all,n,[start-end]"):

The following skeleton is a CMake file used to compile code on the cluster if one wants to carry a piece of code from the workstation to the clusters.

```
#--------------------------------------------------------------------------
# Check if cmake has the required version
#--------------------------------------------------------------------------
cmake_minimum_required(VERSION 2.8.12)
PROJECT(@exe@ LANGUAGES C CXX)
if(COMMAND cmake_policy)
    cmake_policy(SET CMP0003 NEW)
    cmake_policy(SET CMP0002 OLD)
endif(COMMAND cmake_policy)

#----------------------------------------------------------------------------
# CMAKE_MODULE_PATH is used to:
#  -define where are located the .cmake(which contains functions and macros)
#  -define where are the external libraries or modules, third party()
#----------------------------------------------------------------------------
list(APPEND CMAKE_MODULE_PATH @workingDir@/CODE) # DON'T TOUCH

#--------------------------------------------------------------------------
# Compiler Generic Information for all projects
#--------------------------------------------------------------------------
set(CMAKE_VERBOSE_MAKEFILE FALSE)

if (CMAKE_COMPILER_IS_GNUCXX)
 set(CMAKE_C_FLAGS_DEBUG "-g -ggdb -pg -fsanitize=undefined")
 set(CMAKE_C_FLAGS_RELEASE "-O2")
 set(CMAKE_CXX_FLAGS_DEBUG ${CMAKE_C_FLAGS_DEBUG})
 set(CMAKE_CXX_FLAGS_RELEASE ${CMAKE_C_FLAGS_RELEASE})
endif ()
set(CMAKE_BUILD_TYPE RELEASE)

add_library(@exe@_compiler_flags INTERFACE)
target_compile_features(@exe@_compiler_flags INTERFACE cxx_std_11)
```

```
set(gcc_like_cxx "$<COMPILE_LANG_AND_ID:CXX,ARMClang,AppleClang,Clang,GNU>")
target_compile_options(@exe@_compiler_flags INTERFACE
  "$<${gcc_like_cxx}:$<BUILD_INTERFACE:-Wall;-Wextra;-Wshadow;-Wformat=2;-Wunused>>"
)

#-----------------------------------------------------------------------------
# Build shared libs ( if on libraries must be on remote )
#-----------------------------------------------------------------------------
option(BUILD_SHARED_LIBS "Build using shared libraries" @sharedLibs@) #ON/OFF {default off}
#-----------------------------------------------------------------------------
# Set OUTPUT PATH , it will be the working dir using URANIE TLAUNCHERREMOTE
# DO NOT CHANGE workingDir KEYWORD since it retrieves the info from TLauncherRemote->run();
# IF not changed, this will be sourceDirectory ( where you launch root)  +/URANIE/JobName/
# Folders will make will be build and bin,
#-----------------------------------------------------------------------------
set(CMAKE_BINARY_DIR @workingDir@)
set(EXECUTABLE_OUTPUT_PATH ${CMAKE_BINARY_DIR}/bin)
set(LIBRARY_OUTPUT_PATH ${CMAKE_BINARY_DIR}/lib)
#-----------------------------------------------------------------------------
# Set Libraries
#-----------------------------------------------------------------------------


# SET External LIBRARIES
#-----------------------------------------------------------------------------
# Uranie cmake libraries are located on sources folder
#-----------------------------------------------------------------------------
# ROOT
#include_directories(${ROOT_INCLUDE_DIR} ${INCLUDE_DIRECTORIES})
#URANIE
#URANIE_USE_PACKAGE(@workingDir@)
#URANIE_INCLUDE_DIRECTORIES(${LIBXML2_INCLUDE_DIR}
#                         ${ICONV_INCLUDE_DIR_WIN}
#   )

@addExternalLibrary@

# User libraries
@addLibrary@

# add the executable
add_executable(@exe@  ${PROJECT_SOURCE_DIR}/CODE/@exe@)

#Link targets with their libs or libs with others libs
@addDependency@
```

IV.4.5.3 Full script exemple

The following piece o code shows an exemple of submitting script using the `TLauncherByStepRemote` class.

```
#include "TLauncherByStepRemote.h"
#include <libssh/libssh.h>
#include <libssh/sftp.h>
#include <stdlib.h>     //for using the function sleep
{

    // ==================================================================
    // ========================= Classical code =========================
    // ==================================================================
    //Number of samples
    Int_t nS = 30;

    // Define the DataServer
    TDataServer *tds = new TDataServer("tdsflowrate", "DataBase flowrate");
    // Add the eight attributes of the study with uniform law
    tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
    tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
    tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
    tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
    tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
    tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
    tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
    tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

    // Handle a file with flags, usual method
    TString sFileName = TString("flowrate_input_with_flags.in");
    tds->getAttribute("rw")->setFileFlag(sFileName, "@Rw@");
    tds->getAttribute("r")->setFileFlag(sFileName, "@R@");
    tds->getAttribute("tu")->setFileFlag(sFileName, "@Tu@");
    tds->getAttribute("tl")->setFileFlag(sFileName, "@Tl@");
    tds->getAttribute("hu")->setFileFlag(sFileName, "@Hu@");
    tds->getAttribute("hl")->setFileFlag(sFileName, "@Hl@");
    tds->getAttribute("l")->setFileFlag(sFileName, "@L@");
    tds->getAttribute("kw")->setFileFlag(sFileName, "@Kw@");

    // Create a basic doe
    TSampling *sampling = new TSampling(tds, "lhs", nS);
    sampling->generateSample();

    // Define the code with command line....
    TCode *mycode = new TCode(tds, "flowrate -s -f ");

    // ... and output attribute and file
    TAttribute * tyhat = new TAttribute("yhat");
    tyhat->setDefaultValue(-200.0);
    fout->addAttribute(tyhat);
    TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
    mycode->addOutputFile( fout );

    // ==================================================================
    // ================== Specific remote code ========================
    // ==================================================================

    // Create the remote launcher, the third argument is the type of cluster to be used.
    TLauncherByStepRemote *tlch = new TLauncherByStepRemote(tds, mycode, ↩
        TLauncherByStepRemote::EDistrib::kSLURM);

    // Provide the name of the cluster
    tlch->getCluster()->setCluster("marenostrum4");
    // Define the header file
    tlch->getCluster()->setOutputHeaderName("a1.sh");
    tlch->getCluster()->selectHeader("mn4_slurm_skeleton.in");

    tlch->getCluster()->setRemotePath("./multiTestSingle");
    tlch->getCluster()->addClusterDirective("filename"," multiTestSingle");
    tlch->getCluster()->addClusterDirective("queue","debug");
    tlch->getCluster()->addClusterDirective("project","[...]"); //Earth
    tlch->getCluster()->addClusterDirective("numProcs","1");
    tlch->getCluster()->addClusterDirective("wallclock","5:00");
    tlch->getCluster()->setNumberOfCores(1);
    tlch->getCluster()->addClusterDirective("configEnv","/home/[...]");
    // Define the username and authentification method chosen
    tlch->getCluster()->setClusterUserAuth("login","public_key");
    //END OF SLURM SCRIPT
    tlch->setClean();
    tlch->setSave();
    tlch->setVarDraw("hu:hl","yhat>0","");
    /////////////////////
    tlch->setNumberOfChunks(5);
    tlch->setJobName("job3_tlch1");

    tlch->run();
    //tlch->run("nsplit=[0-1]");
    //tlch->run("nsplit=0");
    //tlch->run("nsplit=1");
    //tlch->run("nsplit=[2-3]");
    //tlch->run("nsplit=[3-5]");
    //tlch->run("nsplit=all");
    //tlch->run(); //simultaneous,nonblocking
    tlch->postTreatment();
    //SOME RESULTS

    tds->exportData("_output_flowrate_withRow_.dat","rw:r:tu:tl:hu:hl:l:kw:yhat");
    tds->draw("rw:r:tu:tl:hu:hl:l:kw:yhat","","para");

}
```

Chapter V

The Modeler module

V.1 Introduction

The Modeler module discusses the generation of *surrogate model*s which aim to provide a simpler, and hence faster, model in order to emulate the specified output of a more complex model (and generally time and memory consuming) as a function of its inputs and parameters, provided through a `TDataServer`. The input dataset can either be an existing set of elements (provided by someone else, resulting from simulations or experiments) or it can be a design-of-experiments generated on purpose, for the sake of the ongoing study. The meta-model generation is encoded in Uranie with several different kinds of *surrogate model*, and also different kinds of possible output format. Once created, the resulting model can indeed be transmitted to another code and re-used within or without Uranie, in order to avoid regeneration but also to keep track of achieved performances, as these models can sometimes be created based on a certain randomness (as discussed in the few sections below).

There are several predefined surrogate-models proposed in the Uranie platform:

- The linear regression, discussed in Section V.2
- The chaos polynomial expansion, discussed in Section V.3
- The artificial neural networks, discussed in Section V.5
- The Kriging method, or gaussian process, discussed in Section V.6

It is recommended to follow the law of parsimony (also called Ockham's razor) meaning that the simplest model should be tested first, unless one has insight that it is not well suited for the problem under consideration.

The process of creating a *surrogate model* within Uranie can be summarised in four simple steps and sketched accordingly as shown in Figure V.1. Most of these models are also inheriting from the `TModeler` class, meaning that they will have common methods and options listed below:

Summary: Common method to main classes

- constructors. There are two types (for most of the following models) of constructor:
  - `Model`(**TDataServer***, **TString**, **Option_t***): the architecture is a specific form for all model to condensate information. Usually it contains input names separated by ":", output name and sometime more information. The "," is used, as a separator, to split between fields. Options are discussed below.
  - `Model`(**TDataServer***, **const char***, **const char***, **Option_t***): the input and output name fields are separated. The possible other arguments have to be passed either through the option field, or through specific setters.
  - Sometimes, a constructor is written to take a model from a PMML file.
- options: to be passed as last argument in the constructor, there are two options at this level
  - `nostoreyhat`**:** the results of the modelling is not stored in the `TDataServer` (it is by default).
  - `nointercept`**:** no intercept will be added (only for regressions, for which there is one by default).
- methods: setters and getters. Among the most important one, there is:
  - `getR2`**:** return the R^2 value
  - `getParameters`**:** return a `TVectorD` (a ROOT-class) of the parameters' value.

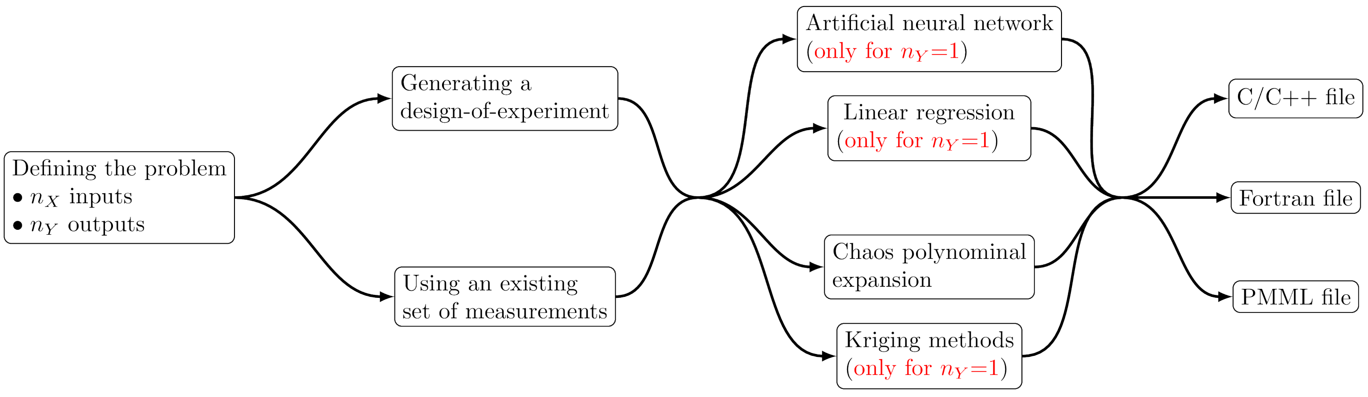

Defining the problem
• n_X inputs
• n_Y outputs
Generating a design-of-experiment
Using an existing set of measurements
Artificial neural network (only for n_Y=1)
Linear regression (only for n_Y=1)
Chaos polynominal expansion
Kriging methods (only for n_Y=1)
C/C++ file
Fortran file
PMML file

Figure V.1: Simplified decomposition of the model creation process into a four important-step recipe.

As shown in Figure V.1, all these *surrogate model*s need to have a set of elements, that can be a design-of-experiments possibly made specifically for this analysis, or a set of measurements or calculations completely independent of Uranie. This set of elements is used as input to create the model that can then be dumped in different format to be re-used within Uranie or by any other code. It is important to enforce that only the C format can be charge, in ROOT directly via the command:

```
gROOT->LoadMacro("file_name.C");
```

The Fortran format cannot be loaded in Uranie, but for some surrogate model it is possible to load the PMML file in the model and then export it in C file, Uranie can load.

Example:

```
TANNModeler* tann = new TANNModeler(tds, "rw:r:tu:tl:hu:hl:l:kw,3,yhat");
....
tann->exportFunction("pmml", "file_name.pmml", "function_name","new");
```

```
....
TANNModeler* tannPMML = new TANNModeler(tds,"file_name.pmml","function_name");
tannPMML->exportFunction("c++", "new_file_name","function_name");
gROOT->LoadMacro("new_file_name.C");
```

The table below summarises which type of export can be used with which class of the modeler module.

	c++	fortran	pmml
TAnisp	X	-	-
TANNModeler	X	X	X
TGLM	X	-	-
TKernel	X	-	-
TkNN	X	X	-
TKriging	X	-	X
TLinearRegression	X	X	X
TPolynomialChaos	X	-	-
TPolynomialRegression	X	X	-

Table V.1: Type of export allowed for different classes

V.2 The TLinearRegression class

When using the TLinearRegression class, one assumes that there is only one output variable and at least one input variable. The data from the training database, shown in Figure V.1, are stored here in a matrix $A(n_S, n_X)$ where n_S is the number of elements in the set and n_X is the number of input variables to be used. The idea is to write any output as $y = \sum_{i=1}^{p} \beta_i h_i$, where β are the regression coefficients and h_i, are the regressors: p simple functions depending on one or more input variables[1] that will be the new basis for the linear regression. A classical simple case is to have $p = n_X$ and $\{h_i(\mathbf{x}) = x_i\}_{i=1,\ldots,n_X}$. The chosen regressors are precised during the construction of the TLinearRegression object, as it takes the TDataServer as first input, a string encoding the regressors to be used and a string encoding the output name.

As a result, a vector of parameters is computed and used to re-estimate the output parameter value. Few quality criteria are also computed, such as R^2 and the adjusted one R^2_{adj} (the value of R^2 tends to increase when additional variables are added to the regression equation even if these variables do not significantly improve the regression, this is why the adjusted version, R^2_{adj} has been created, see [30] for a discussion on these criteria).

Here is an usage-example of the TLinearRegression class:

```
{
  TDataServer * tds = new TDataServer();
  tds->fileDataRead("flowrate_sampler_launcher_500.dat"); // Read the database

  TLinearRegression *tlin = new TLinearRegression(tds, "rw:r:tu:tl:hu:hl:l:kw", "yhat"); // ↩
       Create the linear regression
  tlin->estimate(); // Estimate the parameters

  cout << " ** R2[" << tlin->getR2() << "] R2A[" << tlin->getR2Adjusted() << "] QR2[" <<  ↩
      tlin->getQ2() << "]" << endl;

  tlin->exportFunction("c++", "myASCIIFile", "myFunction");
  }
```

[1]technically, one can also choose 1 as a regressor: this would bring a constant term in the regression.

It results to this output:

```
** R2[0.948985] R2A[0.948154] QR2[0.946835]
```

Summary: `TLinearRegression` object

On top of the methods introduced in the summary block of Section V.1, this class contains:

- **Double_t** `getR2Adjusted`():
  Returns the adjusted value of the R^2 coefficient.

V.3 Chaos polynomial expansion

The basic idea of chaos polynomial expansion (later referred to as *PC*) is that any square-integrable function can be written as $f(\mathbf{x}) = \sum_\alpha f_\alpha \Psi_\alpha(\mathbf{x})$ where $\{f_\alpha\}$ are the PC coefficients, $\{\Psi_\alpha\}$ is the orthogonal polynomial-basis. The index over which the sum is done, α, corresponds to a multi-index whose dimension is equal to the dimension of vector $\mathbf{x}$ (*i.e.* n_X) and whose L1 norm ($|\alpha|_1 = \sum_{i=1}^{n_X} \alpha_i$) is the degree of the resulting polynomial. More theoretical discussion can be found on this in [30].

In Uranie only a certain number of laws can be used and they have their own preferred orthogonal polynomial-basis. The association of the law and the polynomial basis is show in Table V.2. This association is not compulsory, as one can project any kind of law on a polynomial basis which will not be the best-suited one, but the very interesting interpretation of the coefficients will then not be meaningful anymore.

Distribution \ Polynomial type	**Legendre**	**Hermite**	**Laguerre**	**Jacobi**
Uniform	X			
LogUniform	X			
Normal		X		
LogNormal		X		
Exponential			X	
Beta				X

Table V.2: List of best adapted polynomial-basis to develop the corresponding stochastic law

Presentation of the test cases

The `Ishigami` function is usually used as a "benchmark" in the domain of sensitivity because we are able to calculate the exact values of the sensitivity index (discussed in Chapter VI and in [30]). This function is defined by the following equation:

$$f(x_1,x_2,x_3) = \sin x_1 + A\sin^2 x_2 + B x_3^4 \sin x_1$$

EQUATION V.1: Ishigami function

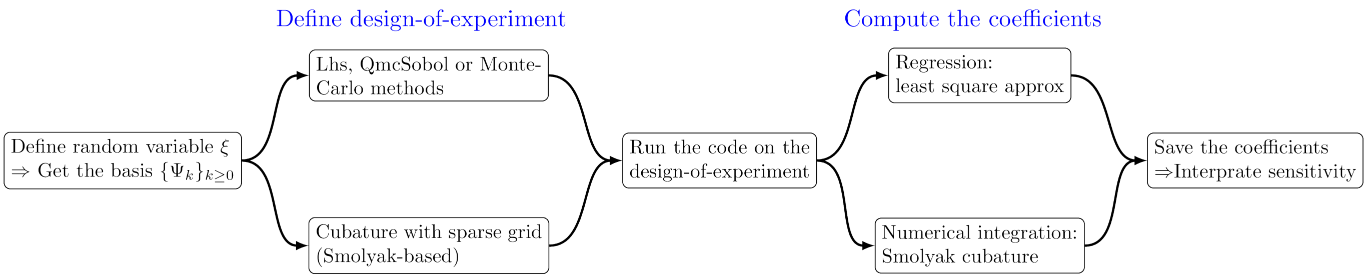
where the x_i follow a uniform distribution on $[-\pi,\pi]$, and A, B are constants. We take A=7, B=0.1 (as done in [41]).

V.3.1 Nisp in a nutshell

The wrapper of the Nisp library, Nisp standing for *Non-Intrusive Spectral Projection*, is a tool allowing to access to Nisp functionality from the Uranie platform. The main features are detailed below.

The Nisp library [10] uses spectral methods based on polynomial chaos in order to provide a surrogate model and allow the propagation of uncertainties if they arise in the numerical models. The steps of this kind of analysis, using the Nisp methodology are represented schematically in Figure V.2 and are introduced below:

- Specification of the uncertain parameters $\mathrm{x_i}$,
- Building stochastic variables associated $\mathrm{x_i}$,
- Building a design-of-experiments
- Building a polynomial chaos, either with a regression or an integration method (see Section V.3.1.1 and Section V.3.1.1)
- Uncertainty and sensitivity analysis



Define design-of-experiment
Compute the coefficients
Define random variable ξ $\Rightarrow$ Get the basis $\{\Psi_k\}_{k\geq 0}$
Lhs, QmcSobol or Monte-Carlo methods
Cubature with sparse grid (Smolyak-based)
Run the code on the design-of-experiment
Regression: least square approx
Numerical integration: Smolyak cubature
Save the coefficients $\Rightarrow$Interprate sensitivity

Figure V.2: Schematic view of the Nisp methodology

V.3.1.1 The regression method

The regression method is simply based on a least-squares approximation: once the design-of-experiments is done, the vector of output $\mathbf{y}(n_S)$ is computed with the code. By writing the correspondence matrix $H(n_S, p)$ and the coefficient-vector β, this estimation is just a minimisation of $||\mathbf{y} - H\beta||^2$. As already stated in Section V.2, this leads to write the general form of the solution as $\beta = (H^T H)^{-1} H^T \mathbf{y}$ which also shows that the way the design-of-experiments is performed can be optimised depending on the case under study (and might be of the utmost importance in some rare case).

In order to perform this estimation, it is mandatory to have more points in the design-of-experiments than the number of coefficient to be estimated (in principle, following the rule $n_S \geq 1.5 \times N_{\rm coeff}$ leads to a safe estimation). For more information on this method, see [30].

V.3.1.2 The integration method

The integration method relies on a more "complex" design-of-experiments. It is indeed recommended to have dedicated design-of-experiments, made with a Smolyak-based algorithms (as the ones cited in Figure V.2). These design-of-experiments are sparse-grids and usually have a smaller number of points than the regularly-tensorised approaches. In this case, the number of samples has not to be specified by the user. Instead, the argument requested describes the level of the design-of-experiments (which is closely intricated, as the higher the level is, the larger the number of samples is). Once this is done, the calculation is performed as a numerical integration by quadrature methods, which requires a large number of computations.

In the case of Smolyak algorithm, this number can be expressed by the number of dimensions n_X and the requested level l as $N_d = 2^l \times l^{n_X - 1}$ which shows an improvement with respect to the regular tensorised formula for quadrature ($\sim 2^{l.n_X}$).

V.3.2 Step 1: Specification of the uncertain parameters

First, it is necessary to build a dataserver containing the uncertain parameters x_i. These parameters are represented by random variables which follow one of these statistical laws given in Table V.2.[2]

Example:

Within the framework of our case-test example, we are going to build a TDS containing 3 attributes which follow a uniform law on the interval $[-\pi, \pi]$.

```
TDataServer *tds = new TDataServer("tdsishigami", "Ex. Ishigami");
tds->addAttribute( new TUniformDistribution("x1", -1*TMath::Pi(), TMath::Pi()));
tds->addAttribute( new TUniformDistribution("x2", -1*TMath::Pi(), TMath::Pi()));
tds->addAttribute( new TUniformDistribution("x3", -1*TMath::Pi(), TMath::Pi()));
```

V.3.3 Step 2: Building stochastic variables

Now, it is necessary to build the stochastic variables ξ_i associated to the uncertain parameters specified above. For that, we create a `TNisp` object from a `TDataServer` object by using the constructor **`TNisp(TDataServer *tds)`**. This constructor builds automatically attributes representing the stochastic variables from the information contained in the tds.

Explanation:

Families of orthogonal polynomials are automatically associated to the attributes representing the uncertain parameters x_i. Given the type of expected law, the corresponding orthogonal polynomial family will be used (as already spotted in Table V.2). However, Uranie gives the possibility to change the family of orthogonal polynomials associated by default to an attribute. It is then necessary to modify, if needed, at the `TNisp` call level, the type of polynomial (by using `getStochasticBasis` of the `TDataServer` library) in order to have the stochastic variables ξ_i represented by attributes following the chosen law.

Example:

In our example, we do not modify the type of the orthogonal polynomials. Then we call, at once, the constructor `TNisp`.

```
// Define of TNisp object
TNisp * nisp = new TNisp(tds);
```

The stochastic variables are represented by 3 internal attributes following a uniform law on [0,1]. A print of the `TNisp` object contents, by using the method **`printLog`**, gives the listing below:

```
** TNisp::printLog[]
   Number of simulation :1
   Number of variable   :3
   Level                :0
```

[2]The followings laws are under development: Triangular and LogTriangular laws.

```
   Stochastic variables:
     -->psi_x1: Uniforme 0 1
        -->psi_x2: Uniforme 0 1
        -->psi_x3: Uniforme 0 1
        Uncertain variables:
        -->x1: Uniforme -3.14159 3.14159
        -->x2: Uniforme -3.14159 3.14159
        -->x3: Uniforme -3.14159 3.14159
fin of TNisp::printLog[]
```

V.3.4 Step 3: Constitution of the sample

Now, it is necessary to generate and evaluate the sample.

V.3.4.1 Generation of the sample

There are two different ways to generate the sample:

- Using the `generateSample` functionality of the library Nisp
- Using the Sampler library of Uranie

V.3.4.1.1 By Nisp library

The Nisp library permits to build a sample thanks to the **`generateSample(TString method,Int_t n)`** method. The parameter *n* represents the level or the size of the sampler depending on the chosen methodology. The parameter *method* represents the building methodology of the sampler. The methods offered by Nisp are the following:

Name of the method	Size	Level
Lhs	X	
Quadrature		X
MonteCarlo	X	
SmolyakFejer		X
SmolyakTrapeze		X
SmolyakGauss		X
SmolyakClenchawCurtis		X
QmcSobol	X	
Petras		X

Table V.3: Methods of sampler generation

Example:

Within the framework of our test case, we build a sampler of type Petras of level 8.

```
Int_t level = 8;
nisp->generateSample("Petras",level);
```

V.3.4.1.2 Using the Sampler library

To build a sample using the Sampler library, see Chapter III.

Once the sample has been built, it is necessary to transfer it to the TNisp object. To do that, we use the **setSample(TString "method",Option_t *option)** method where the parameter *method* is the type of the method used by the library Sampler and the optional parameter *option* is the keyword *"savegvx"* to use if one wants to save the ξ_i in the TDataServer.

Below is an example using the functionality:

```
Int_t nombre_simulations = 100;
TSampling *fsampling = new TSampling(tds, "lhs", nombre_simulations);
fsampling->generateSample();

nisp->setSample("Lhs");
```

V.3.4.2 Evaluation of the sampler

We evaluate the sample using the Uranie Launcher library

```
// Realisation du plan d'experiences
TLauncherFunction * tlf = new TLauncherFunction(tds, "FctIshigami","","Ishigami");
tlf->setDrawProgressBar(kFALSE);
tlf->run();
```

V.3.5 Step 4: Building the polynomial chaos

Given that we have the set (uncertain parameter, stochastic variables and sampler), we can build a TPolynomialChaos object by means of the constructor **TPolynomialChaos(TDataServer *tds, TNisp *nisp)**.

```
// build a polynomial chaos
TPolynomialChaos * pc = new TPolynomialChaos(tds,nisp);
```

One can calculate the coefficients of these polynomials with the **computeChaosExpansion(TString method)** method where the parameter *method* is either the keyword **Integration** or the keyword **Regression**.

Example:

Within the framework of our example, we use the **"Integration"** method.

```
Int_t degree = 8;
pc->setDegree(degree);
pc->computeChaosExpansion("Integration");
```

V.3.6 Step 5: Uncertainty and sensitivity analysis

Now, we can do the uncertainty and sensitivity analyses. With the Nisp library, we have access to these following functionality:

- Mean: **getMean**(y_i) where parameter y_i is either the name or the index of the output variable,
- SVariance: **getVariance**(y_i) where parameter y_i is either the name or the index of the output variables,

- Co-variance: **getCovariance**(y_{1_i}, y_{2_i}) where parameters y_{1_i} and y_{2_i} are either the names or the indexes of the output variables,
- Correlation: **getCorrelation**(y_{1_i}, y_{2_i}) where parameters y_{1_i} and y_{2_i} are either the names or the indexes of the output variable,
- Index of the first order sensitivity **getIndexFirstOrder**(x_i, y_i) where parameter x_i is the name or the index of the input variable and the parameter y_i is the name or the index of the output variable,
- Index of total sensitivity: **getIndexTotalOrder**(x_i, y_i) where parameter x_i is the name or the index of the input variable and the parameter y_i is the name or the index of the output variable,

Example:

Within the framework of our example, the instructions are the following:

```
// Uncertainty and sensitivity analysis
cout << "Variable Ishigami ================" << endl;
cout << "Mean     = " << pc->getMean("Ishigami") << endl;
cout << "Variance = " << pc->getVariance("Ishigami") << endl;
cout << "First Order[1] = " << pc->getIndexFirstOrder("x1","Ishigami") << endl;
cout << "First Order[2] = " << pc->getIndexFirstOrder("x2","Ishigami") << endl;
cout << "First Order[3] = " << pc->getIndexFirstOrder("x3","Ishigami") << endl;
cout << "First Order[1] = " << pc->getIndexTotalOrder("x1","Ishigami") << endl;
cout << "First Order[2] = " << pc->getIndexTotalOrder("x2","Ishigami") << endl;
cout << "First Order[3] = " << pc->getIndexTotalOrder("x3","Ishigami") << endl;
```

The following lines are obtained in the terminal:

```
Variable Ishigami ================
Mean     = 3.5
Variance = 13.8406
First Order[1] = 0.313997
First Order[2] = 0.442386
First Order[3] = 6.50043e-07
Total Order[1] = 0.557568
Total Order[2] = 0.442477
Total Order[3] = 0.24357
```

V.3.7 Other functionalities

There are other Nisp functionalities accessible from Uranie:

Auto-determination of the degree

Recently a new possibility has been introduced in order to be more efficient and less model dependant **when considering the regression procedure**. It is indeed possible to ask for an automatic determination of the best polynomial degree possible. In order to do so, the maximum polynomial degree allowed is computed using the rule of thumb defined above, and the method `computeChaosExpansion` is called with the option "AutoDegree". A polynomial chaos expansion is done from a minimum degree value (one per default, unless otherwise specified thanks to the `setAutoDegreeBoundaries` method) up to the maximum allowed value. To compare the results one to another and be able to decide which one is the best, a Leave-One-Out method (LOO, which consists in the prediction of a value for y_i using the rest of the known values in the training basis, *i.e.* $y_1,\ldots,y_{i-1},y_{i+1},\ldots,y_{n_S}$ for $i = 1\ldots n_S$) and the total *Mean Square Error* (MSE) are used as estimator. These notions are better introduced in [30]. The best estimated degree is evaluated and the polynomial chaos expansion is re-computed, but all results (LOO, MSE...) are stored for all the considered degrees.

This possibility comes with a bunch of new function implemented to modify the behaviour by default. The basic configuration is to start with a degree one polynomial expansion and to scan up to a certain degree $p_{\max}$ that would still satisfy this rule of thumb

$$N_{\text{Coeff}} = \frac{(n_X + p_{\max})!}{n_X! \times p_{\max}!} \leq \frac{n_S}{\chi}$$

χ being a normalisation factor chosen to be 1.5. The usable methods are listed below:

- `setAutoDegreeFactor(double autodeg)`: change the value of the χ factor for the auto normalisation (not recommended). The provided number should be greater than one.
- `setAutoDegreeBoundaries(int amin, int amax)`: change the degree boundaries from one to amin and, if amax is provided, from the level determined by the automatic procedure described above to the chosen amax value. If amax is greater than the automatic determined limit, the default is not changed.

Save data

It is possible to save polynomial chaos data in a *f.dat* file with the **`save(TString sfile)`** method where sfile is the name of the data file. This file will contains the following data:

- stochastic dimension,
- type of the orthogonal polynomial (Legendre, Laguerre or Hermite),
- the degree of the polynomial chaos,
- the number of polynomial chaos coefficients,
- the number of output,
- a list of the coefficient values where the first element is the the mean of the output.

```
// Save the polynomial chaos
pc->save("NispIshigami.dat");
```

Here is a file *.dat* corresponding to our example:

```
nx= 3 Legendre Legendre Legendre no= 8 p= 164 ny= 1 Coefficients[1]= 3.500000e+00 1.625418e ↩
    +00 2.962074e-17
-2.589041e-17 1.575324e-15 5.233520e-17 -3.175950e-18 -5.947228e-01 -1.278829e-17 1.354792e ↩
    -15 -1.290638e+00
-1.858390e-18 5.947865e-18 4.838275e-16 8.497435e-18 1.372419e+00 -1.722653e-19 1.330604e ↩
    -17 -1.575058e-17
5.159489e-18 -1.604235e-15 -1.386085e-17 7.823196e-18 -5.101110e-16 1.268601e-17 1.724496e ↩
    -17 -1.606472e-17
3.412272e-18 1.035117e-17 1.366688e-18 -1.952291e+00 1.603873e-17 1.960259e-16 9.769466e-18 ↩
     -5.051477e-16
1.949287e-01 5.167621e-17 2.218591e-17 -3.783569e-17 1.636383e-17 -1.089702e+00 2.653437e ↩
    -18 1.820189e-17
-3.300252e-17 -1.109147e-17 2.613944e-17 -3.982929e-17 -2.644820e-16 -4.711621e-18 4.091730 ↩
    e-01 5.633658e-17
3.207523e-17 8.469377e-18 -1.623212e-17 -5.221323e-18 -1.348604e-17 -2.549941e-03 -1.380621 ↩
    e-17 4.408383e-18
2.216698e-16 1.055488e-17 -5.971197e-16 -1.077924e-17 -7.500053e-18 -1.398875e-18 8.323369e ↩
    -18 -1.828664e-16
-1.891202e-17 2.846158e-17 1.240056e-18 -7.184406e-17 -4.183157e-17 1.999887e-17 -1.078061e ↩
    -17 1.320948e-18
```

```
4.955143e-20 6.740688e-18 1.357408e+00 -1.599643e-17 -3.404927e-16 -3.826895e-18 -1.184481e ↩
    -16 1.175915e-17
-2.549941e-03 -1.266838e-02 4.550685e-17 1.601655e-17 1.855980e-16 1.171108e-17 1.632677e ↩
    -01 3.035872e-17
-4.384243e-18 -3.127246e-17 1.740356e-17 -3.455894e-19 3.372673e-18 -3.556473e-16 -9.162355 ↩
    e-18 -3.247428e-01
4.190780e-17 -1.547995e-17 -5.808814e-17 9.922144e-18 2.618348e-17 7.213333e-18 -5.300321e ↩
    -06 -4.983518e-18
-3.503280e-16 -6.190117e-18 3.713020e-16 1.065906e-17 -5.448912e-06 3.825624e-18 1.359954e ↩
    -17 -7.719858e-18
-1.803757e-18 4.289915e-17 -1.214645e-18 5.479922e-17 -3.330110e-18 1.579483e-03 2.670695e ↩
    -18 9.359714e-19
2.512771e-02 2.635458e-17 2.622223e-16 -1.309301e-18 8.084506e-18 -1.189488e-17 -4.330879e ↩
    -18 1.247133e-15
-1.863642e-17 2.096038e-16 -2.331035e-18 3.034432e-16 -1.363808e-17 -3.762944e-18 -1.524500 ↩
    e-17 -8.487270e-19
-1.420178e-17 -4.964460e-18 5.812476e-16 2.572312e-17 -2.088466e-17 1.795371e-17 2.011599e ↩
    -16 1.878423e-17
1.608204e-15 2.092277e-17 1.776101e-17 -5.185917e-17 -3.704922e-18 -7.070417e-17 2.010136e ↩
    -17 2.640413e-17
9.647070e-18 -3.393909e-01 -2.530162e-17 8.354699e-16 -1.223963e-18 6.813105e-16 -1.721674e ↩
    -17 2.512771e-02
9.392854e-18 1.579483e-03
```

Saving the macro

It is possible to save the problem in a C macro with the **`exportFunction(char *sfile,char *sfunction)`** method where sfile is the name of the .C file and sfunction is the name of the macro.

```
//save the pv in a program (C langage)
pc->exportFunction("NispIshigami","NispIshigami");
```

Here is the *NispIshigami.C* file corresponding to our example:

```
#include <math.h>

double NispIshigami_beta[1][165]={
3.5,1.62542,2.96207e-17,-2.58904e-17,1.57532e-15,5.23352e-17,-3.17595e-18,-0.594723,
-1.27883e-17,1.35479e-15,-1.29064,-1.85839e-18,5.94787e-18,4.83828e-16,8.49743e-18,1.37242,
-1.72265e-19,1.3306e-17,-1.57506e-17,5.15949e-18,-1.60424e-15,-1.38608e-17,7.8232e-18,
-5.10111e-16,1.2686e-17,1.7245e-17,-1.60647e-17,3.41227e-18,1.03512e-17,1.36669e-18,
-1.95229,1.60387e-17,1.96026e-16,9.76947e-18,-5.05148e-16,0.194929,5.16762e-17,2.21859e-17,
-3.78357e-17,1.63638e-17,-1.0897,2.65344e-18,1.82019e-17,-3.30025e-17,-1.10915e-17,
2.61394e-17,-3.98293e-17,-2.64482e-16,-4.71162e-18,0.409173,5.63366e-17,3.20752e-17,
8.46938e-18,-1.62321e-17,-5.22132e-18,-1.3486e-17,-0.00254994,-1.38062e-17,4.40838e-18,
2.2167e-16,1.05549e-17,-5.9712e-16,-1.07792e-17,-7.50005e-18,-1.39887e-18,8.32337e-18,
-1.82866e-16,-1.8912e-17,2.84616e-17,1.24006e-18,-7.18441e-17,-4.18316e-17,1.99989e-17,
-1.07806e-17,1.32095e-18,4.95514e-20,6.74069e-18,1.35741,-1.59964e-17,-3.40493e-16,
-3.82689e-18,-1.18448e-16,1.17591e-17,-0.00254994,-0.0126684,4.55068e-17,1.60165e-17,
1.85598e-16,1.17111e-17,0.163268,3.03587e-17,-4.38424e-18,-3.12725e-17,1.74036e-17,
-3.45589e-19,3.37267e-18,-3.55647e-16,-9.16236e-18,-0.324743,4.19078e-17,-1.548e-17,
-5.80881e-17,9.92214e-18,2.61835e-17,7.21333e-18,-5.30032e-06,-4.98352e-18,-3.50328e-16,
-6.19012e-18,3.71302e-16,1.06591e-17,-5.44891e-06,3.82562e-18,1.35995e-17,-7.71986e-18,
-1.80376e-18,4.28991e-17,-1.21465e-18,5.47992e-17,-3.33011e-18,0.00157948,2.67069e-18,
9.35971e-19,0.0251277,2.63546e-17,2.62222e-16,-1.3093e-18,8.08451e-18,-1.18949e-17,
-4.33088e-18,1.24713e-15,-1.86364e-17,2.09604e-16,-2.33103e-18,3.03443e-16,-1.36381e-17,
-3.76294e-18,-1.5245e-17,-8.48727e-19,-1.42018e-17,-4.96446e-18,5.81248e-16,2.57231e-17,
-2.08847e-17,1.79537e-17,2.0116e-16,1.87842e-17,1.6082e-15,2.09228e-17,1.7761e-17,
-5.18592e-17,-3.70492e-18,-7.07042e-17,2.01014e-17,2.64041e-17,9.64707e-18,-0.339391,
-2.53016e-17,8.3547e-16,-1.22396e-18,6.81311e-16,-1.72167e-17,0.0251277,9.39285e-18,
0.00157948
};
```

```
int NispIshigami_indmul[165][3]={
0,0,0,1,0,0,0,1,0,0,0,1,2,0,0,1,1,0,1,0,1,0,2,0,0,1,1,0,0,2,3,0,0,2,1,0,2,0,1,1,2,0,1,1,1,
1,0,2,0,3,0,0,2,1,0,1,2,0,0,3,4,0,0,3,1,0,3,0,1,2,2,0,2,1,1,2,0,2,1,3,0,1,2,1,1,1,2,1,0,3,
0,4,0,0,3,1,0,2,2,0,1,3,0,0,4,5,0,0,4,1,0,4,0,1,3,2,0,3,1,1,3,0,2,2,3,0,2,2,1,2,1,2,2,0,3,
1,4,0,1,3,1,1,2,2,1,1,3,1,0,4,0,5,0,0,4,1,0,3,2,0,2,3,0,1,4,0,0,5,6,0,0,5,1,0,5,0,1,4,2,0,
4,1,1,4,0,2,3,3,0,3,2,1,3,1,2,3,0,3,2,4,0,2,3,1,2,2,2,2,1,3,2,0,4,1,5,0,1,4,1,1,3,2,1,2,3,
1,1,4,1,0,5,0,6,0,0,5,1,0,4,2,0,3,3,0,2,4,0,1,5,0,0,6,7,0,0,6,1,0,6,0,1,5,2,0,5,1,1,5,0,2,
4,3,0,4,2,1,4,1,2,4,0,3,3,4,0,3,3,1,3,2,2,3,1,3,3,0,4,2,5,0,2,4,1,2,3,2,2,2,3,2,1,4,2,0,5,
1,6,0,1,5,1,1,4,2,1,3,3,1,2,4,1,1,5,1,0,6,0,7,0,0,6,1,0,5,2,0,4,3,0,3,4,0,2,5,0,1,6,0,0,7,
8,0,0,7,1,0,7,0,1,6,2,0,6,1,1,6,0,2,5,3,0,5,2,1,5,1,2,5,0,3,4,4,0,4,3,1,4,2,2,4,1,3,4,0,4,
3,5,0,3,4,1,3,3,2,3,2,3,3,1,4,3,0,5,2,6,0,2,5,1,2,4,2,2,3,3,2,2,4,2,1,5,2,0,6,1,7,0,1,6,1,
1,5,2,1,4,3,1,3,4,1,2,5,1,1,6,1,0,7,0,8,0,0,7,1,0,6,2,0,5,3,0,4,4,0,3,5,0,2,6,0,1,7,0,0,8
};
void NispIshigami_legendre(double *phi,double x, int no) {
   int i;
   x=2.*x-1.;
   phi[0]=1.; if(no>0)  phi[1]=x;
   for(i=1;i<no;i++)    phi[i+1]= ((2.*i+1.) * x * phi[i] - i * phi[i-1]) / (i+1.);
   for(i=0;i<=no;i++)   phi[i]  = phi[i] * sqrt(2.* i + 1.);
}
void NispIshigami(double *x, double *y)
{
  //////////////////////////////
  //
  //      *********************************************
  //      ** Uranie v2.3/1
  //      ** Export Nisp :
  //      ** Date : Thu Jun 16 15:54:02 2011
  //      *********************************************
  //
  //
  //      *********************************************
  //      ** TDataServer : tdsishigami **
  //      **
  //      ** Ex. Ishigami **
  //      *********************************************
  //
  //
  //////////////////////////////
   int i,j,k,nx,ny,no,p;
   nx=3;ny=1;no=8;p=165;
   double psi[165],phi[3][9],xi[3],s;
   for(i=0;i<nx;i++) xi[i]=x[i];
   NispIshigami_legendre(phi[0],xi[0],8);
   NispIshigami_legendre(phi[1],xi[1],8);
   NispIshigami_legendre(phi[2],xi[2],8);
   for(k=0;k<=p;k++) {
      for(psi[k]=1.,i=0;i<nx;i++) psi[k]=psi[k]*phi[i][NispIshigami_indmul[k][i]];
   }
   for(j=0;j<ny;j++) {
      for(s=0.,k=0;k<=p;k++) s+=NispIshigami_beta[j][k]*psi[k];
      y[j]=s;
   }
}
```

Index of sensitivity of a group of input variables

It is possible to compute the sensitivity index for a group of input variable x_i for a given y_i output variable.

```
cout << "value indice: " <<  pc->getIndex("x1:x3","Ishigami") << endl;
```

Within the framework of our example, this results in the following message:

```
value indice: 0.557523
```

Index of sensitivity of a group of input variables in interaction

It is possible to compute the sensitivity index for a group of input variable x_i in interaction for a given y_i output variable.

```
cout << "value indice: " <<  pc->getIndexInteraction("x1:x3","Ishigami") << endl;
```

Within the framework of our example, this leads to the following line in the terminal:

```
value indice: 0.243525
```

Recover the various dimensions

It is possible to recover:

- the number of input variables with **`getDimensionInput()`**,
- the number of output variables with **`getDimensionOutput()`**
- the number of coefficients with **`getDimensionExpansion()`**

```
cout << "nx = "<< pc->getDimensionInput()  << endl;
cout << "ny = "<< pc->getDimensionOutput() << endl;
cout << "p = "<< pc->getDimensionExpansion() << endl;
```

With our example, this leads to the following line in the terminal:

```
nx = 3
ny = 2
p = 165
```

Calculate output values

It is possible to compute the output of a polynomial chaos for a given x_i input vector.

```
xi[0]=0.8;
xi[1]=0.2;
xi[2]=0.7;
pc->computeOutput(xi);
cout << "Ishigami     = " << pc->getOutput(0) << endl;
```

With our example, this leads to the following line in the terminal:

```
Ishigami = 7.80751
```

Summary: `TNisp` object

- **TNisp**(**`TDataServer`***):

  Construct a `TNisp` object from the dataserver.

- `generateSample`(**TString** type, **int** n, **Option_t** * option)

  Generate the sample following the method `type` (amongst "Lhs", "QmcSobol", "Quadrature", "Petras", "Smolyak-Gauss", "SmolyakTrapeze", "SmolyakFejer", "SmolyakClenshawCurtis"), with a given number of points n (or a level depending on chosen type). Options are introduced in Section V.1.

V.4 Adaptive development in polynomial chaos: the Anisp method

The Wrapper of the Anisp library is a tool allowing to access to Anisp functionalities from Uranie platform. At present, the main features are detailed below.

The Anisp library models uncertainties. It allows to use spectral methods based on polynomial chaos in modelling and propagation studies and adaptive numerical integration if uncertainties arise in the numerical models. Concretely, the Anisp library uses adaptive numerical integration to compute an adapted polynomial expansion with less simulations and a smaller polynomial basis. The steps of an uncertainty analysis by using the Anisp methodology are the following:

- Specification of the uncertain parameters x_i,
- Creation of the `TAnisp` object and specification of the Anisp parameters (tolerance, maximum number of simulations, ...),
- The building of stochastic variables associated ξ_i, the building of an adaptive sampling and building of an adapted chaos polynomial are automatically handled by the Anisp method,
- Uncertainty and sensitivity analysis.

The functionality of Anisp, accessible from Uranie, are explained following a scenario based on an usual example, the Ishigami test case. This scenario is decomposed according to the steps described below.

V.4.1 Step 1: Specification of the uncertain parameters

First, it is necessary to build a dataserver containing the uncertain parameters x_i. These parameters are represented by random variables which follow one of these statistical laws:

- Uniform law,
- Normal law.

For other laws, it is necessary to use `TAttributeFormula` and to configure them from Uniform and Normal laws.

Example 1:

Within the framework of our case-test example, we are going to build a `TDataServer` containing 3 attributes which follow a uniform law on the interval [-π,+π].

```
TDataServer *tds = new TDataServer("tdsishigami", "Ex. Ishigami");
tds->addAttribute( new TUniformDistribution("x1", -1*TMath::Pi(), TMath::Pi()));
tds->addAttribute( new TUniformDistribution("x2", -1*TMath::Pi(), TMath::Pi()));
tds->addAttribute( new TUniformDistribution("x3", -1*TMath::Pi(), TMath::Pi()));
```

Example 2 (using a `TAttributeFormula`):

We are going to build a `TDataServer` containing 3 attributes, the first 2 follow a uniform law on the interval [-1/2,+1/2] and the third is a `TAttributeFormula`. As a consequence x_3 follows Log-Uniform on the interval [exp(-1/2),exp(+1/2)].

```
TDataServer *tdsVenise = new TDataServer("tdsVenise","Ex. Venise");
tdsVenise->addAttribute( new TUniformDistribution("x1", -0.5, 0.5));
tdsVenise->addAttribute( new TUniformDistribution("x2", -0.5, 0.5));
tdsVenise->addAttribute( new TAttributeFormula("x3","TMath::Exp(x2)"));
```

Warning
If a **`TAttributeFormula`** uses x_2 to configure x_3 then x_2 must not be an argument of the `TCode` or the `TLauncherFunction` used to launch computations.

V.4.2 Step 2: Creation of the `TAnisp` Object

Now, we create the `TAnisp` object by using one of the two constructors: **`TAnisp(TDataServer* ,const char *, TString, TString, Option_t *)`** and **`TAnisp(TDataServer* ,TCode *, Option_t *)`**. The first constructor is when we want to approximate a function (see `TLauncherFunction`) and the second when it's a numerical code (see `TCode` and `TLauncher`).

Example:

In our example, we use the first constructor because *Ishigami* is an analytic function. The first `TString` argument indicates the names and order (according to the `TDataServer` attributes) of the input arguments for the function and the second `TString` the name of the output argument:

```
//Definition of the TAnisp object
TAnisp * tanisp = new TAnisp(tds, "FctIshigami","x1:x2:x3","Ishigami");
```

Warning
There must be only one output argument, this is one of the limitation of the Anisp method.

Now, we can specify some of the parameters of the Anisp method, if the defaults are not adapted to the case. The parameters we can specify are:

- the tolerance of the integration algorithm: it determines the precision we want;
- the maximum number of simulations: it stops the computations if it is exceeded;
- the minimum level of interactions (kmin): it specifies the minimum level of interaction investigated, it must be inferior to the number of variables and it increases the initialisation number of simulations of the algorithm;
- the greater index: maximum value in a dimension of a multi-index integration, it's value is limited to 6 in the case of a Normal variable;
- the maximum number of multi-index integration: it is a memory criterion, it's default value is 5000;
- the graphical output parameter: to set or unset the console and graphical outputs;
- the maximum degree: it specify the maximum degree in a dimension for the polynomials;
- the root of the names of the Anisp files: the Anisp library created several files, the user can specify the root of their names, "fichierAnisp_" is the default value.

For each of these parameters there is a method to set the value, in order: `setTolerance`, `setNumberMaxOfSimulati`
`setKMin`, `setGreaterIndice`, `setMaxIndices`, `setlog`, `setDegreeMax` and `setRootFilename`.

There is also three methods which allow to set several parameters:

- `setAllAnispParameters` modifies all the parameters;
- `setAnispParameters` modifies only some parameters: tolerance, kmin, number maximum of simulation, greater index, maximum indices and the graphical outputs parameter;
- `setLightAnispParameters` modifies only the computation parameters: tolerance, kmin and number maximum of simulations.

Example:

```
//Tolerance of the algorithm
Double_t tol = 1.0E-2;
//Maximum number of simulations
Int_t max = 200;
//minimum level of interactions
Int_t kmin = 1;
//setting these parameters
tanisp->setLightAnispParameters(tol,kmin,max);

//root name
TString racine = TString("IshigamiAnisp_");
tanisp->setRootFilename(racine);

// set graphical outputs
tanisp->setlog(kTRUE);
```

Summary: `TAnisp` object

- `TAnisp(`**`TDataServer`** *tds, **const char*** f, **TString** sin="", **TString** sout="", **Option_t** *option="")

  Create a `TAnisp` object for a function `f`, inputs and outputs being specified respectively in `sin` and `sout`.

- `TAnisp(`**`TDataServer`** *tds, **void*** f(**double** *,**double** *), **const char*** fname, **TString** sin="", **TString** sout="", **Option_t** *option="")

  Create a `TAnisp` object for a function `f`, whose name is fname, inputs and outputs being specified respectively in `sin` and `sout`.

- `TAnisp(`**`TDataServer`** *tds, **TCode** * mycode, **Option_t** * option="")

  Create a `TAnisp` object for the external code `mycode`.

- `setTolerance(`**Double_t** tol)

  Set the tolerance.

- `setKMin(`**Int_t** kmin)

  Set the minimum level of interaction.

- `setNumberMaxOfSimulations(`**Int_t** nmos)

  Set the maximum number of simulations.

- `setGreaterIndice(`**Int_t** lev)

  Set the indice maximum value.

- `setMaxIndices(`**Int_t** mi)

  Set the maximum number of indices.

- `setlog(`**Bool_t** mylog)

  Set the graphical output parameter.

- `setDegreeMax(`**Int_t** deg)

  Set the maximum degree.

- `setRootFilename(`**TString** name)

  Set the root name.

- `setAllAnispParameters(`**double** t,**int** km,**int** nmos, **int** lev,**int** mi,**bool** mylog,**int** deg, **TString** name)

  Set all the Anisp parameters

- `setAnispParameters(`**Double_t** tol,**Int_t** kmin,**Int_t** nmos, **Int_t** lev,**Int_t** mi,**Bool_t** mylog)

  Set some Anisp parameters.

- `setAnispParameters(`**Double_t** tol,**Int_t** kmin,**Int_t** nmos)

  Set computations parameters of Anisp.

V.4.3 Step 3: Running the Anisp method

The construction of the associated stochastic variables ξ_i, the generation of the adaptive sampling and the adapted polynomial chaos expansion are automatically handled by the Anisp method. Once the `TAnisp` object is created and new parameters set (if necessary), the user is left with the `runAnisp` method of the class `TAnisp`.

Example:

```
//run of Anisp
tanisp->runAnisp();
```

After using the `runAnisp` method, it's possible for the user to set new parameters and then restart the algorithm where it has stopped by using the method `restartAnisp` of the class `TAnisp`.

Example:

```
//new parameters
tol = 1.0E-6; max = 1000; kmin = 2;
tanisp->setLightAnispParameters(tol,kmin,max);
//restart of Anisp
tanisp->restartAnisp();
```

Summary: running of the `Anisp` method

- `runAnisp`()

  Launch the Anisp method.

- `restartAnisp`()

  Launch the Anisp method from the last iteration of a previous run.

V.4.4 Step 4: Uncertainty and sensitivity analysis

Now, we can do the uncertainty and sensitivity analysis. With the Anisp library, we have access to the functionality of the Nisp library.

Indeed the `TAnisp` object create a `TPolynomialChaos` object during the call of the `runAnisp` and `restartAnisp` methods. The user can then use the `getTPolynomialChaos` method to get back the `TPolynomialChaos` object and perform the uncertainty and sensitivity analysis by using the dedicated method of this class.

Another way is to use the exportFunction method which creates a C++ file containing the polynomial under the form of a C++ function. The user can then perform on it any statistical analysis.

Example:

```
//getting the TPolynomialChaos object
TPolynomialChaos * pc;
TPolynomialChaos P1(tanisp->getTPolynomialChaos());
pc = &P1;

//Uncertainty and sensitivity analysis using TPolynomialChaos functionality
cout<<endl;
cout<<"Mean = "<< pc->getMean(0)<< endl;
cout<<"Variance = "<< pc->getVariance(0)<< endl;
cout<<endl;
```

```
cout<<"First Order[1] = "<< pc->getIndexFirstOrder(0,0)<<endl;
cout<<"First Order[2] = "<< pc->getIndexFirstOrder(1,0)<<endl;
cout<<"First Order[3] = "<< pc->getIndexFirstOrder(2,0)<<endl;
cout<<endl;
cout<<"Total Order[1] = "<< pc->getIndexTotalOrder(0,0)<<endl;
cout<<"Total Order[2] = "<< pc->getIndexTotalOrder(1,0)<<endl;
cout<<"Total Order[3] = "<< pc->getIndexTotalOrder(2,0)<<endl;
cout<<endl;
cout<<"Ordered functionnal ANOVA"<<endl;
Double_t seuil = 0.98;
pc->getAnovaOrdered(seuil,0);

// Export of the polynomial in a C++ file
tanisp->exportFunction("c++","AnispIshigami.C","AnispIshigami");
```

Summary: Uncertainty and sensitivity analysis

- `getTPolynomialChaos`()

  Get the `TPolynomialChaos` object.

- `exportFunction`(**const char** *lang, **const char** *file="", **const char** *name="", **Option_t** *option="")

  Export the polynomial in a C++ file.

V.5 The artificial neural network

Warning
This *surrogate model*, as implemented in Uranie, requires the Opt++ prerequisite (as discussed in Section I.1.2.2).

The Artificial Neural Networks *(ANN)* in Uranie are Multi Layer Perceptron *(MLP)* with one or more hidden layer (containing n_H^i neurons, where i is use to identify the layer) and one or more output attribute. We can export them in ASCII file as *"C"*, *"Fortran"* and *"PMML"* formats to reuse them later on within Uranie or not.

V.5.1 The working principle

The artificial neural networks done within Uranie need input from **OPT++** and can also benefit from the computation power of *graphical process unit* (GPU) if available. Their implementation is done through the `TANNModeler` Uranie-class, and their conception and working flow is detailed in three steps in the following part, summarised in Figure V.3. For a thoroughly description of the artificial neural network, see [30]

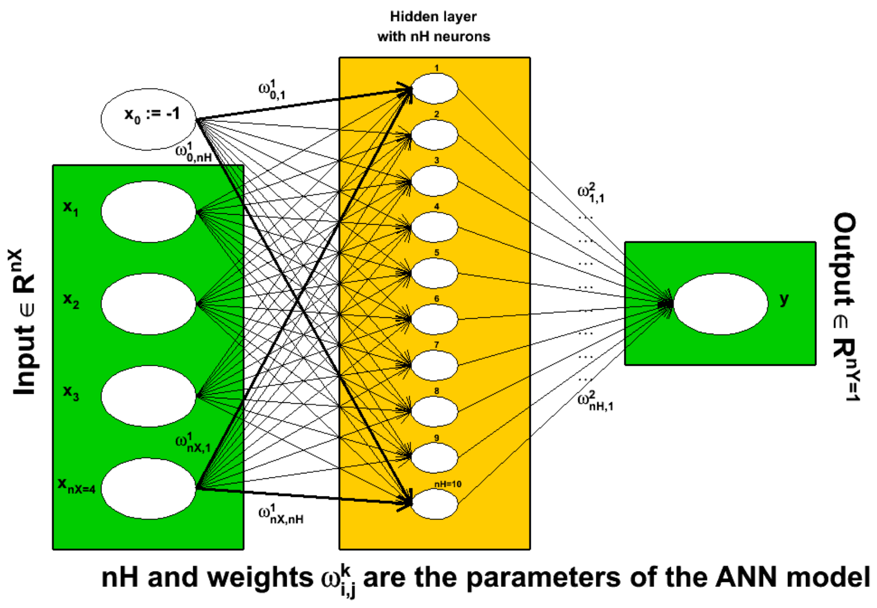


Figure V.3: Schematic description of the working flow of an artificial neural network as used in Uranie

The first step is the creation of the artificial neural network in Uranie; there are several compulsory information that should be given at this stage:

- a pointer to the `TDataServer` object
- the input variables to be used (as for the linear regression, it is perfectly possible to restrain to a certain number of inputs)
- the number of neurons in the hidden layer
- the name of the output variable

The three last information are gathered in a single string, using commas to separate clearly the different parts. This is further discussed in Section V.5.2.

The second step is the training of the ANN. Every formal neuron is a model that does not talk to any other neuron on the considered hidden layer, and that is characterised by (taking, for illustration purpose, an index $j \in [1, n_H]$):

- the weight vector that affects it, $w^1_{0,j} \dots w^1_{n_X,j}$, using the Figure V.3 (the superscript 1 stands for the layer index, as there is only one hidden layer in Uranie implementation).
- an activation function θ that goes along with the way the inputs and output are normalised.

Combining these with the inputs give the internal state of the considered neuron,

$$s = \theta(w_0 + \sum_{i=1}^{n_X} w_i x_i)$$

that is function of the weights' value, which are estimated from the training. To perform it, one can specify the tolerance parameter to stop the learning process (the default value being 1e-06). It is however necessary to give the number of times (a random permutation of) the test base will be presented for training and the number of times the ANN is trained (from random start weights) with a given permutation of the test database. The training session ends, keeping the best performance model obtained. This is further discussed in Section V.5.3 (and in [30]).

Finally, the constructed neural network can be (and should be) exported. Different format are available to allow the user to plug the resulting function in its code whether it is using Uranie or not.

Warning It is recommended to save the best estimated model, as running twice the same code will not give the same results. There is indeed a stochastic part in the splitting of the training database that will induce differences from one run to another.

This is further discussed in Section V.5.4.

V.5.2 Constructor

The `TANNModeler` constructor is specified with a `TDataServer` which contains the input attributes and the output attribute and with an integer to define the number of hidden neurons in the hidden layer. All these information are stored in a string. These string is the second argument of the constructor.

```
TDataServer * tds = new TDataServer("tds","my TDS");
tds->fileDataRead("flowrate_sampler_launcher_500.dat");

TANNModeler* tann = new TANNModeler(tds, "rw:r:tu:tl:hu:hl:l:kw,4,yhat");
```

In case one wants to use several hidden layer, the number of hidden neurons in each layer has to be specified in the architecture string, separated by commas. For example with three layers of 2, 3 and 4 neurons (dummy example for illustration purpose only), one would write something like this:

```
TANNModeler* tannML = new TANNModeler(tds, "rw:r:tu:tl:hu:hl:l:kw,2,3,4,yhat");
```

To split the data of the `TDataServer` in two databases, learning and test, we specify either a proportion (real value between 0.0 and 1.0) of patterns or the number of patterns (integer greater than 2) to build the learning database. The other patterns are stored in the test database. No validation base is explicitly created by Uranie.

Summary: TANNModeler constructors

1. `TANNModeler(` **TDataServer** *tds, **TString** sstruct, **Double_t** dratio = 0.80)

   Constructs on `TModeler` object from a `TDataServer` with the structure *"sstruct"* which contains the input attributes (*"x1:x2:x3"*), the number of hidden units (*"8"*) (separated by commas if one wants to use more than one hidden layer) and the output attribute (*"y"*) for the following example *"x1:x2:x3,8,y"*. The *dratio* is the percentage (if lesser than 1) or the number (if greater than 1) of patterns to build the learning database.

V.5.3 Training

The training and testing database split is done based on the ratio introduced in the `TANNModeler` constructor, introduced previously. This is further discussed in [30].

```
tann->setNormalization(TANNModeler::kMinusOneOne);
tann->setFcnTol(1e-8);
tann->train(3, 2, "test");
```

Summary: TANNModeler training

1. `setNormalization`(**TANNModeler::ENorm** [TANNModeler::kMinusOneOne | TANNModeler::kCR | TANNModeler::kZeroOne])

   Defines the type of normalisation for the input and output data and the transfer function in the hidden layer, given a chosen enumerator. The correspondence is as follow:

   - **`TANNModeler::kMinusOneOne`**: hyberbolic tangent *("tanh")*
   - **`TANNModeler::kCR`**: sigmoid
   - **`TANNModeler::kZeroOne`**: sigmoid

   The default normalisation is **kCR**, then the transfer function of the hidden neurons is a sigmoid.

2. `setFcnTol`(**Double_t** dTol)

   Specifies the tolerance parameter to stop the learning process for the ANN. The default value is **1e-06**

3. `train`(**Int_t** niter = 10, **Int_t** ninit = 10, **Option_t** *option = "text")

   Launch the training process of the neural network for *niter* iterations which contained *ninit* initialisation each other and with the option *"option"*.

V.5.4 Export

```
tann->exportFunction("c++", "uranie_ann_flowrate","ANNflowrate");
tann->exportFunction("fortran", "uranie_ann_flowrate","ANNflowrate");
tann->exportFunction("pmml", "uranie_ann_flowrate.pmml", "ANNflowrate4","new");
```

Summary: TANNModeler export

1. `exportFunction`(**const char*** lang, **const char*** file="", **const char*** name="", **Option_t** *option = "")

   Export the ANN in an ASCII file with name *"file"* for the language (*C C++/Fortran/PMML*) in a function with name *"name"*. If *file* is empty, the filename will be *"Output name"*_nH*"Number of Hidden units"*.*"lang"*. If the name is empty, then the function is called "Fct_*"Output name"*.

V.6 The kriging method

Warning
This *surrogate model*, as implemented in Uranie, requires the NLopt prerequisite (as discussed in Section I.1.2.2).

First developed for geostatistic needs, the kriging method, named after D. Krige and also called Gaussian Process method (denoted GP hereafter) is another way to construct a *surrogate model*. It recently became popular thanks to a series of interesting features:

- it provides a prediction along with its uncertainty, which can then be used to plan the simulations and therefore improve the predictions of the *surrogate model*
- it relies on relatively simple mathematical principle
- some of its hyper-parameters can be estimated in a Bayesian fashion to take into account *a priori* knowledge.

Kriging is a family of interpolation methods developed in the 1970s for the mining industry [11]. It uses information about the "spatial" correlation between observations to make predictions with a confidence interval at new locations. In order to produce the prediction model, the main task is to produce a spatial correlation model. This is done by choosing a correlation function and search for its optimal set of parameters, based on a specific criterion.

The *gpLib* library [12] provides tools to achieve this task. Based on the gaussian process properties of the kriging [13], the library proposes various optimisation criteria and parameter calculation methods to find the parameters of the correlation function and build the prediction model.

The present chapter describes the integration of the gpLib inside Uranie, and how to create a new kriging model and use it. We first give a quick reminder of what kriging is. Next, we describe how to create a new prediction model using Uranie. Then, we explain how to use this model to predict new observations. Finally, we present some advanced usages of the Uranie/gpLib interface.

The aim of developing a meta-model is to emulate a code or a function which takes a certain number of inputs (that one can write like $\mathbf{x} = (x_1 \ldots x_{n_X}))$ and gives a scalar y as a result. If one has a set of n_S measurements, meaning that the relation $y_i = y(\mathbf{x}^i)$ is perfectly known for $i = 1 \ldots n_S$, the gaussian process model can be used to re-estimate the observations and provide an expected value for new measurements.

V.6.1 Running a kriging

The kriging procedure in Uranie can be schematised in five steps, depicted in Figure V.4 and an example can be found in Section XIV.6.14. As for the polynomial chaos expansion discussed in Section V.3, the classes are mainly interfaces, here to the classes extracted from the `gpLib` library. Here is a brief description of the steps:

- get a training site. Either produced by a design-of-experiments from a model definition, or taken from anywhere else, it is mandatory to get this basis (the larger, the better).
- define the kriging options. This is heavily discussed in Section V.6.2, all the options discussed in the aforementioned section will be inputs of the `TGPBuilder` class that will be build the kriging model.
- set the parameter's values. It can be set by hands, but it is highly recommended to proceed through an optimisation, to get the best possible parameters.
- build the kriging method. This is done with a specific method of `TGPBuilder` which returns a `TKriging` object. This object is the one that would be saved (if requested) and used to perform the next step.
- test the obtained kriging model. This is done by running the kriging model over a new basis (the test one) using a `TLauncher2` object (which is the equivalent of a `TLauncher` but from the **URANIE::Relauncher** module). This module will be discussed later on in Chapter VIII. With this approach, a one-by-one estimation of the prediction is performed. To get the complete prediction with the new location covariance matrix (see [30] for this theory behind this discussion) a different method has been introduced, see Section V.6.3.2.

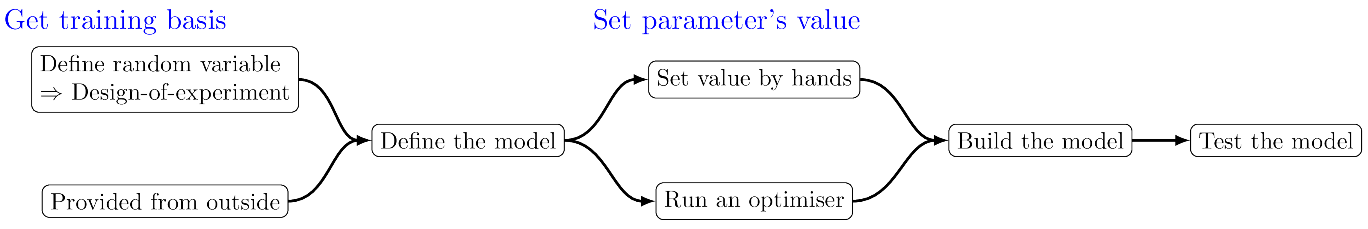


Figure V.4: Schematic description of the kriging procedure as done within Uranie

V.6.2 Construction of a kriging model

V.6.2.1 Description of the main classes

The Uranie wrapper to the gpLib is based on 4 main classes:

- `TGPBuilder`: allow to search for the optimal parameters of the correlation function and to build the kriging model;
- `TCorrelationFunction`: parent class of the correlation functions used to model the spatial correlation of the observations;
- `TGPCostFunction`: parent class of the criteria used to find the optimal parameters of the correlation function;
- `TKriging`: used to predict new values (with confidence interval).

Most of the time, users will only need to use the `TGPBuilder` and `TKriging` classes.

V.6.2.2 Example: construction of a simple Kriging model

The example code below creates a prediction model of the type "simple kriging" (it can be found in Section XIV.6.9).

```
{
  // Load observations
  TDataServer *tdsObs = new TDataServer("tdsObs", "observations");
  tdsObs->fileDataRead("utf_4D_train.dat");

  // Construct the GPBuilder
  TGPBuilder *gpb = new TGPBuilder(tdsObs,            // observations data
           "x1:x2:x3:x4",     // list of input variables
           "y",               // output variable
           "matern3/2");      // name of the correlation function


  // Search for the optimal hyper-parameters
  gpb->findOptimalParameters("ML",            // optimisation criterion
           100,              // screening design size
           "neldermead",     // optimisation algorithm
           500);             // max. number of optimisation iterations

  // Construct the kriging model
  TKriging *krig = gpb->buildGP();

  // Display model information
  krig->printLog();

  }
```

In this script, after loading the observation data into a dataserver, we create a `TGPBuilder` object. To do so, one should provide the observations, the name of the attributes corresponding to the inputs, the name of the attribute corresponding to the output, and the name of the correlation function (here, it is a *Matern 3/2* correlation function. The training (`utf_4D_train.dat`) and testing (`utf_4D_test.dat`) database can both be found in the document folder of the Uranie installation (`${URANIESYS}/share/uranie/docUMENTS`). Please refer to Table V.6 or [30] for the list of available correlation functions).

The next step is to find the optimal parameters of the correlation function. To achieve this, we can use the `TGPBuilder::fi` function. It is a "helper function" where all the tedious work is done automatically. The drawback is that the user cannot modify some properties (parameters range, optimisation precision, etc.), or use some interesting features of Uranie (non-random sampling, evolutionary optimisation algorithms, distributed computing, etc.). Section V.6.4 gives some examples of how to search for optimal parameters without using the function `findOptimalParameters`.

The search for the optimal parameters requires the user to choose:

- an optimisation criterion (in the example: the *Maximum likelihood*);
- the size of the screening design-of-experiments;
- an optimisation algorithm;
- a maximum number of optimisation runs.

The search for optimal parameters will start by the search of a "good" starting point for the optimisation. This is done by evaluating the criterion on a LHS design-of-experiments of the input space. This "screening" procedure is optional and can be skipped by setting the design size to 0.

The optimisation procedure will start either at the "best" location found by the screening, or at a default location. The chosen optimisation algorithm is then used to search for an optimal solution. Depending on various conditions, convergence can be difficult to achieve. The number of optimisation runs is thus limited to 1000 by default, but can be increased or decreased by the user. The list of available optimisation criteria is available in Table V.4. The list of optimisation algorithms is available in Table V.5. You can also refer to the developer documentation.

When the search is finished, and if everything went well, we can create a kriging model using the function `TGPBuilder::bu` This function returns the pointer to a `TKriging` object which can be used for the prediction of new points. The function `TKriging::printLog` shows some of the properties of the model, like the parameters of the correlation function and the leave one out performances (RMSE and Q2):

```
*******************************
** TKriging::printLog[]
*******************************
 Input Variables:      x1:x2:x3:x4
 Output Variable:      y
 Deterministic trend:
 Correlation function: URANIE::Modeler::TMatern32CorrFunction
 Correlation length:   normalised   (not normalised)
                       1.6181e+00 (1.6172e+00 )
                       1.4372e+00 (1.4370e+00 )
                       1.5026e+00 (1.5009e+00 )
                       6.7884e+00 (6.7944e+00 )

 Variance of the gaussian process:      70.8755
 RMSE (by Leave One Out):               0.499108
 Q2:                                    0.849843
*******************************
```

Warning
Internally, the inputs are automatically normalised to the interval [0, 1]. As a consequence, the "normalised" correlation lengths correspond to distances in the normalised space, and the "not normalised" lengths correspond to distances in the original space.

V.6.2.3 Regularisation

Sometimes, the optimisation fails due to an ill-conditioned covariance matrix. The corresponding cost is set to an absurd value and this is signalled, at the end, by a warning message of the form (when XX estimations fail out of the total number TOT):

```
<WARNING> TGPBuilder::Screening procedure. The Cholesky decomposition has failed XX out of ↩
    TOT times </WARNING>
```

It can then be useful to apply a regularisation parameter (mathematically similar to the "nugget effect" of geostatistics).

To apply the regularisation, we simply have to write:

```
gpb->setRegularisation(1e-6);
```

before calling `findOptimalParameters`. Typical values for the regularisation parameter lie between 1e-12 and 1e-6.

V.6.2.4 Deterministic trend and bayesian prior

Defining a deterministic trend is done at the construction of the `TGPBuilder` object. It can be generated automatically by using a keyword ("const" or "linear")

```
// Load observations
TDataServer *tdsObs = new TDataServer("tdsObs", "observations");
tdsObs->fileDataRead("utf_4D_train.dat");

// Construct the GPBuilder
TGPBuilder *gpb = new TGPBuilder(tdsObs,            // observations data
"x1:x2:x3:x4",      // list of input variables
"y",                // output variable
"matern3/2",        // name of the correlation function
"linear");          // trend defined by a keyword
```

or manually by writing a formula where the terms are separated by colon character (":")

```
// Load observations
TDataServer *tdsObs = new TDataServer("tdsObs", "observations");
tdsObs->fileDataRead("utf_4D_train.dat");

// Construct the GPBuilder
TGPBuilder *gpb = new TGPBuilder(tdsObs,            // observations data
"x1:x2:x3:x4",      // list of input variables
"y",                // output variable
"matern3/2",        // name of the correlation function
"1:x1:x2+x4^2");    // custom trend
```

When using the "const" keyword, the trend is a constant value. When using "linear", the trend is a linear combination of all the normalised input variables (plus a constant).

When using a formula, each sub-element must be either a constant or a combination of one or more inputs. It must also respect ROOT's formula syntax (cf. TFormula description for details).

Warning
Custom trend applies on "original space" variables, while automatic trend applies on "normalised space" variables.

This shows the impact of all deterministic trend cases implementation:

- "": no trend given.
- "const": There is an extra parameter β_0 to determine, as $E(y) = \beta_0$.
- "linear": There are $n_X + 1$ extra parameters to be determined, as $E(y) = \beta_0 + \sum_{k=1}^{n_X} \beta_k x_k$.
- "a:b:c": This is customised trend. The given string is split according to ":". There will be as many new parameters as there are entries and "a", "b", and "c" can be functions of various input variables. For instance with a line like $"1 : x_1 : \cos(0.45 \times x_2^2 + 0.38)"$, the trend would be $E(y) = \beta_0 + \beta_1 \times x_1 + \beta_2 \times \cos(0.45 \times x_2^2 + 0.38)$

If we have an *a priori* knowledge on the mean and variance of the trend parameters, we can use it to perform a bayesian study, as in the example below (which can be found in Section XIV.6.11):

```
{
    // Load observations
   TDataServer *tdsObs = new TDataServer("tdsObs", "observations");
   tdsObs->fileDataRead("utf_4D_train.dat");

   // Construct the GPBuilder
   TGPBuilder *gpb = new TGPBuilder(tdsObs,            // observations data
            "x1:x2:x3:x4",      // list of input variables
            "y",                // output variable
            "matern3/2",        // name of the correlation function
            "linear");          // trend defined by a keyword


   // Bayesian study
   Double_t meanPrior[5] = {0.0, 0.0, -1.0, 0.0, -0.1};
   Double_t covPrior[25] = {1e-4, 0.0 , 0.0 , 0.0 , 0.0 ,
          0.0 , 1e-4, 0.0 , 0.0 , 0.0 ,
          0.0 , 0.0 , 1e-4, 0.0 , 0.0 ,
          0.0 , 0.0 , 0.0 , 1e-4, 0.0 ,
          0.0 , 0.0 , 0.0 , 0.0 , 1e-4};
   gpb->setPriorData(meanPrior, covPrior);

   // Search for the optimal hyper-parameters
   gpb->findOptimalParameters("ReML",          // optimisation criterion
            100,              // screening design size
            "neldermead",     // optimisation algorithm
            500);             // max. number of optimisation iterations

   // Construct the kriging model
   TKriging *krig = gpb->buildGP();

   // Display model information
   krig->printLog();

  }
```

Warning
Please note that we change the optimisation criterion from "ML" (Maximum likelihood) to "ReML" (Restricted Maximum likelihood) as the former cannot handle bayesian studies.

The information displayed by the `TKriging::printLog` function shows the values of the 5 trend parameters calculated by the bayesian study. The *a priori* variance of the parameters being very small, we can see that the trend parameters are very close to their *a priori* means.

```
*******************************
** TKriging::printLog[]
*******************************
 Input Variables:      x1:x2:x3:x4
 Output Variable:      y
 Deterministic trend:  linear
 Trend parameters (5): [3.06586494e-05; 1.64887174e-05; -9.99986787e-01; 1.51959859e-05;  ↩
     -9.99877606e-02 ]
 Correlation function: URANIE::Modeler::TMatern32CorrFunction
 Correlation length:   normalised   (not normalised)
                       2.1450e+00 (2.1438e+00 )
                       1.9092e+00 (1.9090e+00 )
                       2.0062e+00 (2.0040e+00 )
                       8.4315e+00 (8.4390e+00 )

 Variance of the gaussian process:      155.533
 RMSE (by Leave One Out):               0.495448
 Q2:                                    0.852037
*******************************
```

Tip
If you want to deactivate the bayesian mode after setting the prior data, call `gpb->setUsePrior(kFALSE)` before searching for the optimal parameters. You can reactivate the bayesian mode later by calling `gpb->setUsePrior(kTRUE)`.

V.6.2.5 Measurement error

If we want to take a measurement error into account when looking for the optimal hyper-parameters, we can do so by calling the function:

```
gpb->setHasMeasurementError(kTRUE);
```

The measurement error is supposed to follow a normal law of mean 0. If the variance *vMes* is unknown, a new parameter *alpha = vMes/vGP* (where *vGP* is the variance of the gaussian process) will be added to the list of hyper-parameters to optimise.

If we know the variance of the measurement error to be a constant, it must be set using the function

```
Double_t vMes = 0.0025;
gpb->setMeasurementErrorVariance(vMes);
```

We may even know the covariance matrix of a complex measurement error. In such case, it must be set using the function

```
Double_t covMes[25] = {25*1e-4, 0.0 ,     0.0 , 0.0 ,     0.0 ,
0.0 , 25*1e-4, 0.0 , 0.0 ,     0.0 ,
0.0 , 0.0 , 25*1e-4, 0.0 ,     0.0 ,
0.0 , 0.0 ,     0.0 , 25*1e-4, 0.0 ,
0.0 , 0.0 ,     0.0 , 0.0 , 25*1e-4};
gpb->setMeasurementErrorCovMatrix(covMes);
```

In both cases, the variance of the gaussian process, which is otherwise determined analytically from the other hyper-parameters, becomes one of the hyper-parameters to optimise.

V.6.2.6 Optimisation options

The optimisation phase of the search for optimal parameters depends on the choice of a criterion and an optimisation algorithm. In the tables below, we present a quick description of the available options. For more details about the

Criteria	keyword	no trend	trend	bayesian	description
Maximum likelihood	ML	yes	yes	no	Look for the hyper-parameters which maximise the density of the gaussian vector of the observation outputs. It cannot be used for bayesian studies.
Restricted Maximum likelihood	ReML	no	yes	yes	Same as ML, but adapted to allow bayesian studies. It cannot be used without a trend.
Leave One Out	Loo	yes	yes	yes	Thanks to the linear nature of the kriging model, the Leave One Out error has an analytic formulation. It can be used as a quality criterion.

Table V.4: Optimisation criteria

algorithms, please consult the NLopt website (http://ab-initio.mit.edu/wiki/index.php/NLopt_Algorithms).

V.6.2.7 Choice of the initial point of the optimisation

In the examples presented so far, the initial point of the optimisation procedure was either a default location or determined via a random sampling of the input space. If a specific location is needed, it can be set as shown below (which can be found in Section XIV.6.10):

```
{
 // Load observations
 TDataServer *tdsObs = new TDataServer("tdsObs", "observations");
 tdsObs->fileDataRead("utf_4D_train.dat");

 // Construct the GPBuilder
 TGPBuilder *gpb = new TGPBuilder(tdsObs, "x1:x2:x3:x4", "y", "matern3/2");

 // Set the correlation function parameters
 Double_t params[4] = {1.0, 0.25, 0.01, 0.3};
 gpb->getCorrFunction()->setParameters(params);

 // Find the optimal parameters
 gpb->findOptimalParameters("ML",          // optimisation criterion
```

[3]the additional "e" is to respect the original author's wish that only *his* implementation of the algorithm be named "Subplex".

Algorithm	**keyword**	**Use gradient**	**description**
Nelder-Mead Simplex	NelderMead	no	This is an implementation of the "simplex" algorithm, simple and quite efficient in most cases. However, it does not always converge to a local minimum
Subplex	Subplexe[3]	no	Another version of the simplex supposed to be more robust and efficient than Nelder-Mead
Constrained Optimisation By Linear Approximations	Cobyla	no	Construct successive linear approximations of the cost function (using a simplex) and optimise them.
Bound Optimisation BY Quadratic Approximation	Bobyqa	no	Construct quadratic approximations of the cost function and optimise them. May perform poorly on cost function that are not twice-differentiable !
Principal Axis	Praxis	no	Use the "principal axis" method to converge to a solution without estimating a gradient. May be slow to converge.
Low-storage BFGS	BFGS	yes	An improved implementation of the BFGS algorithm which reduces memory consumption and convergence time.
Preconditioned truncated Newton	Newton	yes	An implementation of the Newton algorithm which reduces memory consumption and convergence time.
Method of Moving Asymptotic	MMA	yes	Construct local approximation of the cost function based on the gradient, the constraints and a quadratic "penalty" term. The approximation obtained is "easy" to optimise. This algorithm is guaranteed to converge to a local minimum whatever the starting point.
Sequential Least-Squares Quadratic Programming	SLSQP	yes	The algorithm optimises successive second-order approximations of the cost function (via BFGS updates), with first-order (linear) approximations of the constraints. As it uses "true" BFGS, this algorithm becomes slow for large numbers of parameters
Shifted limited-memory variable-metric	VariableMetric	yes	This algorithm is adapted to large scale optimisation problems.

Table V.5: Optimisation algorithm

```
                                0,            // screening size MUST be equal to 0
          "neldermead", // optimisation algorithm
          500,          // max. number of optimisation iterations
          kFALSE);      // we don't reset the parameters of the GP builder

  // Create the kriging model
  TKriging *krig = gpb->buildGP();

  // Display model information
  krig->printLog();

  }
```

In this case, only the initial correlation lengths need to be set, so we simply retrieve the pointer to the correlation function object and use the `setParameters` function. Then, we call `findoptimalParameters` with two particularities:

- the screening size must be set to 0, otherwise the initial point will be defined as the best location of a random sampling;
- the last parameter of the function, called "reset", must be set to `kFALSE`. This insures that the current values of the hyper-parameters are not reset to their default values.

Warning
When setting the correlation lengths manually, remember that the values you provide are considered normalised.

The resulting model is printed as follow:

```
*******************************
** TKriging::printLog[]
*******************************
 Input Variables:      x1:x2:x3:x4
 Output Variable:      y
 Deterministic trend:
 Correlation function: URANIE::Modeler::TMatern32CorrFunction
 Correlation length:   normalised   (not normalised)
                       1.6182e+00 (1.6173e+00 )
                       1.4373e+00 (1.4371e+00 )
                       1.5027e+00 (1.5011e+00 )
                       6.7895e+00 (6.7955e+00 )

 Variance of the gaussian process:      70.8914
 RMSE (by Leave One Out):               0.49911
 Q2:                                    0.849842
*******************************
```

V.6.2.8 Observations update

One situation where it might be interesting to continue an optimisation procedure from the last optimal solution found is when the observation dataset is updated "on the fly". A typical use-case is the iterative construction of a design-of-experiments.

If the contents of the dataserver of the observations are modified, we must update the internal matrices of the `TGPBuilder`, but without modifying the hyper-parameters we have found so far. The function `TGPBuilder::updateOb` does exactly that:

```
{
  // Load observations
  TDataServer *tdsObs = new TDataServer("tdsObs","");
  tdsObs->fileDataRead("utf_4D_train.dat");

  // Construct the GPBuilder
  TGPBuilder *gpb = new TGPBuilder(tdsObs, "x1:x2:x3:x4", "y", "matern3/2");

  // Search for the optimal hyper-parameters
  gpb->findOptimalParameters("ML", 100, "neldermead", 500);

  // Create a new observation
  Double_t newObs[5] = {7.547605e-01, 8.763968e-01, 1.255390e-01, 6.370434e-01, 1.189220e ↩
      +00};

  // Add the new observation to tdsObs
  Double_t newLine[6];
  newLine[0] = tdsObs->getNPatterns()+1; // Index of the new observation
  newLine[1] = newObs[0];
  newLine[2] = newObs[1];
  newLine[3] = newObs[2];
  newLine[4] = newObs[3];
  newLine[5] = newObs[4];

  tdsObs->getTuple()->Fill(newLine);

  // Update the GP Builder
  gpb->updateObservations();

  // Search for the new optimal hyper-parameters, starting from their current values
  gpb->findOptimalParameters("ML", 0, "bobyqa", 500, kFALSE);

  // Create the model
  TKriging *krig = gpb->buildGP();

  // Display model information
  krig->printLog();

  }
```

The resulting model is printed as follow:

```
*******************************
** TKriging::printLog[]
*******************************
 Input Variables:      x1:x2:x3:x4
 Output Variable:      y
 Deterministic trend:
 Correlation function: URANIE::Modeler::TMatern32CorrFunction
 Correlation length:   normalised   (not normalised)
                       1.6272e+00 (1.6263e+00 )
                       1.4443e+00 (1.4441e+00 )
                       1.5095e+00 (1.5078e+00 )
                       6.8387e+00 (6.8447e+00 )

 Variance of the gaussian process:      71.8595
 RMSE (by Leave One Out):               0.498229
 Q2:                                    0.85008
*******************************
```

V.6.3 Usage of a Kriging model

V.6.3.1 Prediction of a new data set, one-by-one approach

In the following example, we use a newly constructed kriging model to estimate the output of a new dataset (the code can be found in Section XIV.6.12):

```
{
  // Load observations
  TDataServer *tdsObs = new TDataServer("tdsObs", "observations");
  tdsObs->fileDataRead("utf_4D_train.dat");

  // Construct the GPBuilder
  TGPBuilder *gpb = new TGPBuilder(tdsObs,            // observations data
           "x1:x2:x3:x4",      // list of input variables
           "y",                // output variable
           "matern3/2");        // name of the correlation function


  // Search for the optimal hyper-parameters
  gpb->findOptimalParameters("ML",            // optimisation criterion
           100,              // screening design size
           "neldermead",     // optimisation algorithm
           500);             // max. number of optimisation iterations

  // Construct the kriging model
  TKriging *krig = gpb->buildGP();

  // Display model information
  krig->printLog();

  // Load the data to estimate
  TDataServer *tdsEstim = new TDataServer("tdsEstim", "estimations");
  tdsEstim->fileDataRead("utf_4D_test.dat");

  // Construction of the launcher
  TLauncher2 *lanceur = new TLauncher2(tdsEstim,          // data to estimate
               krig,               // model used for the estimation
               "x1:x2:x3:x4",      // list of the input variables
               "yEstim:vEstim");   // name given to the model's outputs

  // Launch the estimations
  lanceur->solverLoop();

  // Display some results
  tdsEstim->draw("yEstim:y");

  }
```

The part where the kriging model is constructed is identical to Section V.6.2.2.

After printing the model information, the new data are loaded into a dataserver. The `TKriging` class is a children of `TSimpleEval` and can thus be used by a `TMaster` (cf. Chapter VIII). Here, we create a `TLauncher2` object. It receives the data to estimate, the model to apply, the names of the input variables, and the names of the output variable. The latter is added to the dataserver and receive the responses of the model when the function `solverLoop` is called.

Warning Using the prototype shown above (and recall here) to construct `TLauncher2` instance is perfectly fine as long as one does not want to use constant or temporary attributes. If you don't know what's discussed here, we strongly invite you to have a look here Section VIII.5.1.

```
TLauncher2(TDataServer *tds, TStandardEval *fun, const char *in, const char *out);
```

Even though one can define at once the input and output attributes (that should be in the input `TDataServer`) but there is no way to state that one input attribute is constant to set its value later-on just before the `solverLoop` call. In order to do that, one should stick to the "classical" **Relauncher** architecture:

- add input attributes to the evaluator using the `addInput` and / or `setInputs` methods;
- add output attributes to the evaluator using the `addOutput` and / or `setOutputs` methods;
- create the `TLauncher2` instance with this prototype

  ```
  TLauncher2(TDataServer *tds, TStandardEval *fun);
  ```

For each point of the data set, we now have an estimation of the output ("yEstim") and a variance of this estimation ("vEstim"). We can access these values via the dataserver's visualisation or computation tools.

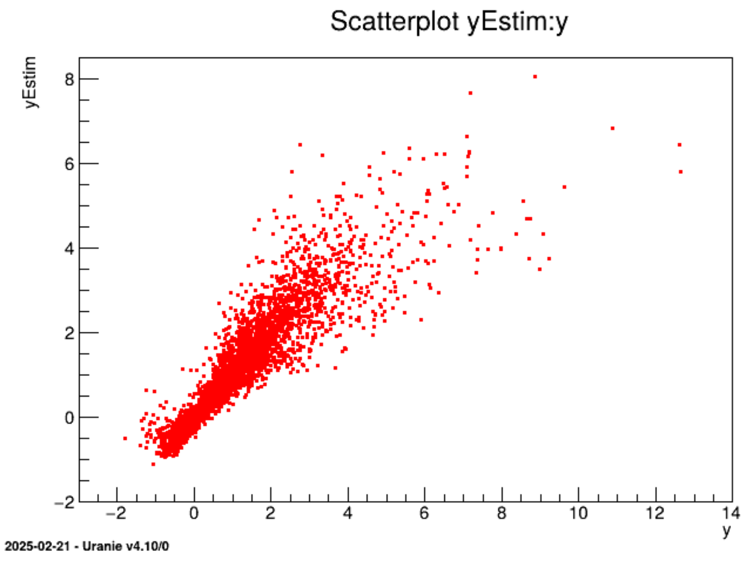


Figure V.5: Estimation using a simple Kriging model

V.6.3.2 Prediction of a new data set, global approach

In the following example, we use a newly constructed kriging model to estimate the output of a new dataset, but unlike the previous case (see Section V.6.3.1) the prediction is done at once, taking into account the covariance of the new locations input to produce the prediction covariance matrix. The code can also be bound in Section XIV.6.13.

```
{
  // Load observations
  TDataServer *tdsObs = new TDataServer("tdsObs", "observations");
  tdsObs->fileDataRead("utf_4D_train.dat");
```

```
  // Construct the GPBuilder
  TGPBuilder *gpb = new TGPBuilder(tdsObs,             // observations data
           "x1:x2:x3:x4",      // list of input variables
           "y",                // output variable
           "matern1/2");        // name of the correlation function


  // Search for the optimal hyper-parameters
  gpb->findOptimalParameters("ML",             // optimisation criterion
           100,             // screening design size
           "neldermead",    // optimisation algorithm
           500);            // max. number of optimisation iterations

  // Construct the kriging model
  TKriging *krig = gpb->buildGP();

  // Display model information
  krig->printLog();

  // Load the data to estimate
  TDataServer *tdsEstim = new TDataServer("tdsEstim", "estimations");
  tdsEstim->fileDataRead("utf_4D_test.dat");

  // Reducing the database to 1000 first event (prediction cov matrix of a million value !)
  int nST=1000;
  tdsEstim->exportData("utf_4D_test_red.dat","",Form("tdsEstim__n__iter__<=%d",nST));
  tdsEstim->fileDataRead("utf_4D_test_red.dat",false,true); // Reload reduce sample

 krig->estimateWithCov(tdsEstim, // data to estimate
                       "x1:x2:x3:x4",// list of the input variables
                       "yEstim:vEstim", // name given to the model's outputs
                       "y", //name of the true reference if validation database
                       ""); //options

  TCanvas *c2=NULL;
  // Residuals plots if true information provided
  if( tdsEstim->isAttribute("_Residuals_") )
  {
      c2 = new TCanvas("c2","c2",1200,800);
      c2->Divide(2,1);
      c2->cd(1);
      // Usual residual considering uncorrated input points
      tdsEstim->Draw("_Residuals_");
      c2->cd(2);
      // Corrected residuals, with prediction covariance matrix
      tdsEstim->Draw("_uncorrResiduals_");
  }

  // Retrieve all the prediction covariance coefficient
  tdsEstim->getTuple()->SetEstimate( nST * nST ); //allocate the correct size
  // Get a pointer to all values
  tdsEstim->getTuple()->Draw("_CovarianceMatrix_","","goff");
  double *cov=tdsEstim->getTuple()->GetV1();

  //Put these in a matrix nicely created
  TMatrixD Cov(nST,nST);
  Cov.Use(0,nST-1,0,nST-1,cov);

  //Print it if size is reasonnable
  if(nST<10)
      Cov.Print();
```

```
}
```

The part where the kriging model is constructed is identical to Section V.6.2.2.

After printing the model information, the new data are loaded into a dataserver. Unlike the previous case (see Section V.6.3.1), the validation database is reduced to a fifth of its original size, as the global approach will have to compute the input covariance matrix but also, from it, the prediction covariance matrix, both of which have, just for the reduced case, a million coefficients. The method to do this is called `estimateWithCov` and it can take 5 parameters:

- a pointer to the dataserver;
- the list of inputs to be used;
- the name of the outputs. Only two are allowed and compulsory: the predicion and its conditional variance;
- the name of the true value of the model (optional). Usable only with validation / test database;
- and the options.

For each point of the data set, we now have an estimation of the output ("yEstim") and a variance of this estimation ("vEstim"). This method also produce few more attributes:

- "_CovarianceMatrix_": this is a strange way to store the final prediction covariance matrix. It is a new vector attribute whose size is constant throughout the dataserver and it is the size of the sample. For every entry it corresponds to the row of the covariance matrix and the recommended way to extract it is the way shown above, *i.e.*

```
// Retrieve all the prediction covariance coefficient
tdsEstim->getTuple()->SetEstimate( nST * nST ); //allocate the correct size
// Get a pointer to all values
tdsEstim->getTuple()->Draw("_CovarianceMatrix_","","goff");
double *cov=tdsEstim->getTuple()->GetV1(); // The data itself

//Put these in a matrix nicely created (high level object to deal with)
TMatrixD Cov(nST,nST);
Cov.Use(0,nST-1,0,nST-1,cov);
```

- "_Residuals_" and "_uncorrResiduals_": available only if the true value of the model is provided along (as fourth parameter of the function prototype). It is basically the following ratio for all locations in the tested database P:

$$\frac{\hat{y}_i - y_i}{\sigma_i},\ i = 1, \ldots, n_P.$$

  It is depicted in our case in Figure V.6.

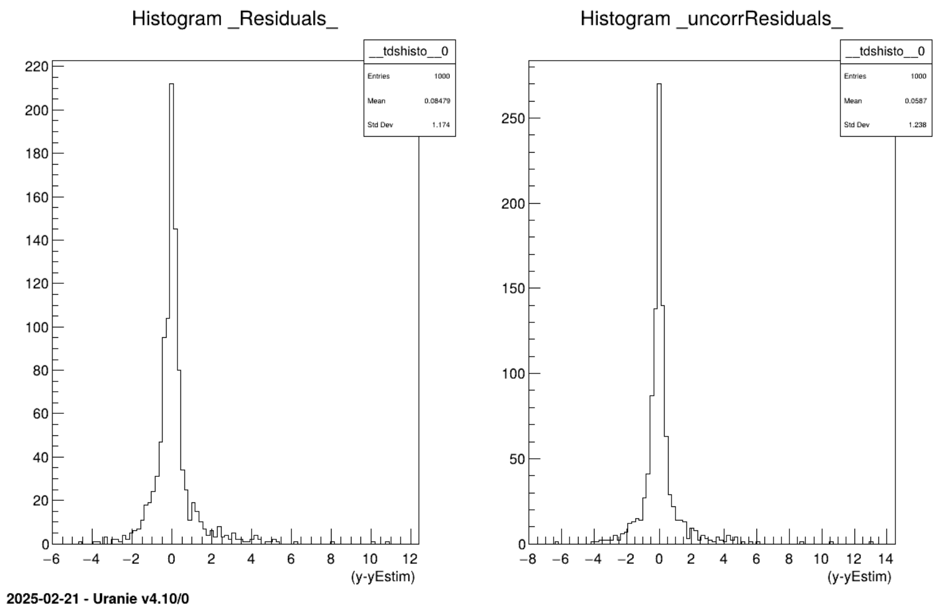


Figure V.6: Residual distribution using a validation database with and without prediction covariance correction.

V.6.3.3 Saving and loading a model

It is not currently possible to export a kriging model as a standalone C or fortran function. There are various reasons for this. The main one is that saving a kriging model means saving several matrices which can be huge when the number of observations and/or the number of input variables is large. It is thus impractical to save them in a text format as is usually done for the other algorithms of `URANIE::Modeler`.

The alternative is to export the information needed to *build* the `TKriging` object with a `TGPBuilder`. This is done by the function `exportGPData`:

```
gpb->findOptimalParameters("ML",100,"neldermead",500);

// export optimal parameters to a file
gpb->exportGPData("modelData.dat");
```

The file created ("modelData.dat") is a standard "Salome table" file, with a special `#TITLE` line in its header:

```
#NAME: tdsKriging
#TITLE: URANIE::Modeler::TMatern32CorrFunction: [1.61744349e+00 1.43718257e+00 1.50122179e ↩
    +00 6.79556828e+00]; variance: 7.09060005e+01;
#DATE: Wed Aug  6 10:24:36 2014
#COLUMN_NAMES: x1| x2| x3| x4| y| tdsKriging__n__iter__

2.186488000e-01 1.534173000e-01 6.196718000e-01 3.452344000e-01 1.235132000e+00 1
9.467816000e-02 3.830201000e-01 5.978722000e-03 7.689013000e-01 2.527613000e+00 2
4.716125000e-01 1.566920000e-01 8.623332000e-01 9.878017000e-01 -6.464496000e-01 3
1.346335000e-01 7.979628000e-02 5.091562000e-01 7.396531000e-01 2.461238000e+00 4
6.238671000e-01 2.995599000e-01 2.385627000e-01 9.548934000e-01 -3.266630000e-03 5
...
```

The `#TITLE` line contains the class name of the correlation function, its parameters, the variance of the gaussian process, and any other parameter required to construct the kriging model. The rest of the file contains the observation inputs and output.

Warning
Do not modify the content of the file unless you know exactly what you are doing. It may produce a model which has nothing to do with the original one, or the file may not be readable at all.

The script below presents how to reconstruct the model from the saved file:

```
{
 // Load the model data file into a data server
 TDataServer *tdsModelData = new TDataServer(); // VERY IMPORTANT: do not give any name or ↩
    title to this tds !
 tdsModelData->fileDataRead("modelData.dat");

 // Create a GP builder based on this data server
 TGPBuilder *gpb = new TGPBuilder(tdsModelData);

 // Create the kriging model
 TKriging *krig = gpb->buildGP();

 // Display model information
 krig->printLog();

  }
```

The constructor of the `TGPBuilder` object takes only the dataserver with the model data. It reads the `#TITLE` line and extracts the information. It is thus not necessary to call `findOptimalParameters`. The function `buildGP` can be used immediately to create the model. This macro is finished by printing the resulting model:

```
*******************************
** TKriging::printLog[]
*******************************
 Input Variables:      x1:x2:x3:x4
 Output Variable:      y
 Deterministic trend:
 Correlation function: URANIE::Modeler::TMatern32CorrFunction
 Correlation length:   normalised   (not normalised)
                       9.7238e-01 (9.7183e-01 )
                       8.5615e-01 (8.5606e-01 )
                       9.4116e-01 (9.4013e-01 )
                       4.3909e+00 (4.3948e+00 )

 Variance of the gaussian process:      18.1787
 RMSE (by Leave One Out):               0.510406
 Q2:                                    0.842968
*******************************
```

V.6.4 Advanced usage

The function `TGPBuilder::findOptimalParameters` greatly simplifies the process of finding good parameters for the correlation function. However, in some cases, it might be interesting to perform this procedure differently. In the following sections, we go "inside the engine" to allow advanced users to go beyond `findOptimalParameters` if they want to.

V.6.4.1 The correlation function

A `TGPBuilder` object contains a `TCorrelationFunction` object which will compute the correlation between the observation points. A clone of this correlation function object is stored in the `TKriging` object, where it will estimate the correlation between the observations and a new point.

It is possible to set the correlation function object manually, either at construction, or later using `TGPBuilder::setCorrFu`
Whether the user creates a new object or simply passes a keyword to the `TGPBuilder`, a new correlation function object will be created inside the `TGPBuilder` and destroyed with it.

The example below illustrates the construction of a `TGPBuilder` by passing a correlation function object:

```
...
// Definition of the initial correlation lengths
Double_t corrLengths[4] = {1.0, 0.25, 0.01, 0.3};

// Construction of the correlation function
TMatern32CorrFunction *corrFunc = new TMatern32CorrFunction(4, corrLengths);

// Construction of the GP builder
TGPBuilder *gpb = new TGPBuilder(tdsObs, "x1:x2:x3:x4", "y", corrFunc);

// The correlation function object is no longer needed. We can destroy it
delete corrFunc;
...
```

The table below presents the list of correlation functions currently available in Uranie:

Class Name	**keyword**	**number of parameters**[4]	**order of the parameters in the *parameters array*** [5]
TExponentialCorrFunction	exponential	2*d	[p1, p2, ..., pd, l1, l2,..., ld]
TGaussianCorrFunction	gauss	d	[l1, l2,..., ld]
TIsotropicGaussianCorrFunction	isogauss	1	[l]
TMaternICorrFunction	maternI	2*d	[v1, v2, ..., vd, l1, l2,..., ld]
TMaternIICorrFunction	maternII	d+1	[v, l1, l2,..., ld]
TMaternIIICorrFunction	maternIII	d+1	[v, l1, l2,..., ld]
TMatern12CorrFunction	matern1/2	d	[l1, l2,..., ld]
TMatern32CorrFunction	matern3/2	d	[l1, l2,..., ld]
TMatern52CorrFunction	matern5/2	d	[l1, l2,..., ld]
TMatern72CorrFunction	matern7/2	d	[l1, l2,..., ld]

Table V.6: Correlation functions

For a more complete description of the available correlation functions (at least a more mathematical one) please report to [30].

V.6.4.2 The cost function

As mentioned before, Uranie provides three criteria (or *cost functions*) to select "good" hyper-parameters for the kriging model. If needed, the user can create his own `TGPCostFunction` object and apply his own searching method to find the optimal hyper-parameters.

The `TGPCostFunction` classes inherit from `TSimpleEval` and can thus be used by a `TMaster` (cf. Chapter VIII). Complete examples of how to use a cost function object are given in the next sections.

[4] *d* is the number of dimensions of the input space
[5] *p* is the power of the exponential; *v* is the regularisation parameter of the Matèrn function; *l* is the correlation length.

V.6.4.3 Example: parameters searched by LHS

The example below illustrate the "manual" search for optimal hyper-parameters through a screening of the parameters space.

```
{
 using namespace URANIE::DataServer; // TDataServer, TLogUniformDistribution
 using namespace URANIE::Modeler;    // TGPBuilder, TGPMLCostFunction, TKriging
 using namespace URANIE::Sampler;    // TBasicSampling
 using namespace URANIE::Relauncher; // TLauncher2

 // Load observations
 TDataServer *tdsObs = new TDataServer("tdsObs","");
 tdsObs->fileDataRead("utf_4D_train.dat");

 // Construct the GPBuilder
 TGPBuilder *gpb = new TGPBuilder(tdsObs, "x1:x2:x3:x4", "y", "matern3/2");

 // construction of the cost function
 TGPMLCostFunction *cost = new TGPMLCostFunction(gpb);

 //----------------------------------//
 // Construction of the LHS sampling //
 //----------------------------------//
 // construction of the attributes
 TLogUniformDistribution *cl1 = new TLogUniformDistribution("cl1", 1e-6, 10.0);
 TLogUniformDistribution *cl2 = new TLogUniformDistribution("cl2", 1e-6, 10.0);
 TLogUniformDistribution *cl3 = new TLogUniformDistribution("cl3", 1e-6, 10.0);
 TLogUniformDistribution *cl4 = new TLogUniformDistribution("cl4", 1e-6, 10.0);

 // Construction of the data server containing the exploration
 // of the correlation lengths' space
 TDataServer *tdsParam = new TDataServer("tdsParam", "");
 tdsParam->addAttribute(cl1);
 tdsParam->addAttribute(cl2);
 tdsParam->addAttribute(cl3);
 tdsParam->addAttribute(cl4);

 // Construction of the sampler and generation of the data
 Int_t screeningSize = 1000;
 TBasicSampling *s = new TBasicSampling(tdsParam, "lhs", screeningSize);
 s->generateSample();

 // Evaluate the cost function on the sampling
 TLauncher2 *l = new TLauncher2(tdsParam,cost,"cl1:cl2:cl3:cl4","cost:varGP");
 l->solverLoop();

 //---------------------------------------------------------//
 // Search for the minimum value of the cost and retrieve the //
 // corresponding correlation lengths                         //
 //---------------------------------------------------------//
 // Retrieve the cost values
 Double_t *costValues = tdsParam->getTuple()->getBranchData("cost");

 // find the index of the minimum cost
 Int_t minIndex = TMath::LocMin(screeningSize, costValues);
 Double_t optimalCost = costValues[minIndex];

 // retrieve the values of the optimal correlation lengths
 Double_t optimalParams[4];
 optimalParams[0] = tdsParam->getValue("cl1", minIndex);
 optimalParams[1] = tdsParam->getValue("cl2", minIndex);
 optimalParams[2] = tdsParam->getValue("cl3", minIndex);
 optimalParams[3] = tdsParam->getValue("cl4", minIndex);

 // retrieve the value of the GP variance
 Double_t optimalVar = tdsParam->getValue("varGP", minIndex);

 //-----------------------------------------------------------//
 // Update the GP Builder with the newly found parameters and //
 // create the model                                          //
 //-----------------------------------------------------------//
 // set the parameters of the correlation function
 gpb->getCorrFunction()->setParameters(optimalParams);

 // set the GP variance
 gpb->setVariance(optimalVar);

 // Create the model
 TKriging *krig = gpb->buildGP();

 // Display model information
 krig->printLog();

  }
```

The first step of the procedure is similar to what we have seen so far: we are loading the observations into a data server and create a `TGPBuilder`. The change occurs when we create a `TGPMLCostFunction` object. It will allow us to evaluate the likelihood of the gaussian process for a given set of hyper-parameters. For this, it needs an access to the observations, the correlation function, etc. This is done by giving it a pointer to the `TGPBuilder` object.

The next step is to create the dataset which will allow us to explore the parameter space. In this example, we only care about the correlation lengths along each of the 4 input variables. We create a `TLogUniformDistribution` object for each of them in order to have a good screening resolution on small correlation lengths. Then, we create the dataserver that will hold the dataset, add the attributes to it, create a sampler object and generate the data (cf. Chapter III for details).

When the dataset is ready, we need to evaluate the cost function on it. An easy way to do it is to create a `TLauncher2` object, and give it the dataset, the cost function object, the input names, and the output names. When the function `solverLoop` is called, the entries of the dataset are sent to the cost function, and the results of the latter are stored in the dataserver.

The next operation is the retrieval of the optimal parameters. We need to identify the point with the minimal cost, and extract the values of the corresponding correlation lengths and GP variance. When done, we need to manually update the `TGPBuilder`. First, we set the new parameters of the correlation function by retrieving the correlation function pointer (using `TGPBuilder::getCorrFunction`), and by calling `TCorrelationFunction::setParameters`. Then, we set the variance of the gaussian process by calling `TGPBuilder::setVariance`.

At this stage, the `TGPBuilder` object is ready to create a kriging model with the new hyper-parameters found. This is what we do by calling `buildGP`.

This example is a simplified version of what is done inside `findOptimalParameters`. But here, we can decide to distribute the computations, use another screening procedure, etc. We can even use another cost function !

The resulting model is printed as follow:

```
*******************************
** TKriging::printLog[]
*******************************
 Input Variables:      x1:x2:x3:x4
 Output Variable:      y
 Deterministic trend:
 Correlation function: URANIE::Modeler::TMatern32CorrFunction
 Correlation length:   normalised   (not normalised)
                       4.4685e-01 (4.4660e-01 )
                       2.3956e-01 (2.3953e-01 )
                       2.9826e-01 (2.9794e-01 )
                       1.6450e+00 (1.6464e+00 )

 Variance of the gaussian process:      1.89625
 RMSE (by Leave One Out):               0.578768
 Q2:                                    0.798086
*******************************
```

V.6.4.4 Example: parameters search by direct optimisation

The example below illustrate the "manual" search for optimal hyper-parameters through a direct optimisation.

```
{
  using namespace URANIE::DataServer; // TDataServer, TLogUniformDistribution
  using namespace URANIE::Modeler;    // TGPBuilder, TGPMLCostFunction, TKriging
  using namespace URANIE::Reoptimizer;// TNlopt, TNloptNelderMead

  // Load observations
  TDataServer *tdsObs = new TDataServer("tdsObs","");
  tdsObs->fileDataRead("utf_4D_train.dat");

  // Construction of the GPBuilder
  TGPBuilder *gpb = new TGPBuilder(tdsObs, "x1:x2:x3:x4", "y", "matern3/2");

  // Construction of the cost function
  TGPMLCostFunction *costFunc = new TGPMLCostFunction(gpb);

  //--------------------------------------------//
  // Construction of the parameters to optimise //
  //--------------------------------------------//
  // construction of the input attributes
  TAttribute *cl1 = new TAttribute("cl1", 1e-6, 10.0);
  TAttribute *cl2 = new TAttribute("cl2", 1e-6, 10.0);
  TAttribute *cl3 = new TAttribute("cl3", 1e-6, 10.0);
  TAttribute *cl4 = new TAttribute("cl4", 1e-6, 10.0);
  // construction of the output attributes
  TAttribute *cost  = new TAttribute("cost");
  TAttribute *varGP = new TAttribute("varGP");

  // Construction of the data server containing the parameters to optimise
  TDataServer *tdsParam = new TDataServer("tdsParam", "");
  tdsParam->addAttribute(cl1);
  tdsParam->addAttribute(cl2);
  tdsParam->addAttribute(cl3);
  tdsParam->addAttribute(cl4);

  // Add the parameters to optimise to the cost function
  costFunc->setInputs(4, cl1, cl2, cl3, cl4);
  costFunc->setOutputs(2, cost, varGP);

  //-------------------------------//
  // Construction of the optimiser //
  //-------------------------------//
  // Choice of an optimisation algorithm
  TNloptNelderMead *solver = new TNloptNelderMead();
// Construction of the optimizer, which receives the data server storing
// the optimal result, the criterion and the optimisation algorithm.
TNlopt *opt = new TNlopt(tdsParam, costFunc, solver);

// Optimisation options
opt->addObjective(cost); // criterion to minimise
vector<double> start{1e-2, 1e-2, 1e-2, 1e-2};
opt->setStartingPoint(start.size(),&start[0]); // starting point for the optimisation
opt->setMaximumEval(1000); // maximum number of evaluation calls
opt->setTolerance(1e-12); // convergence criterion

// Launch the optimisation procedure
opt->solverLoop();

//-----------------------------------------------------------//
// Update the GP Builder with the newly found parameters and //
// create the model                                          //
//-----------------------------------------------------------//
// Retrieve the optimal parameters. In this case, the data server contains only
// one set of parameters, thus we don't have to search for the minimal cost.
Double_t optimalParams[4];
optimalParams[0] = tdsParam->getValue("cl1",0);
optimalParams[1] = tdsParam->getValue("cl2",0);
optimalParams[2] = tdsParam->getValue("cl3",0);
optimalParams[3] = tdsParam->getValue("cl4",0);
// set the parameters of the correlation function
gpb->getCorrFunction()->setParameters(optimalParams);

// Retrieve the computed variance
Double_t optimalVar = tdsParam->getValue("varGP",0);
// Set the new variance of the GP
gpb->setVariance(optimalVar);

// Create the model
TKriging *krig = gpb->buildGP();

// Display model information
krig->printLog();

}
```

The beginning of the script is the same as in the previous example (see Section V.6.4.3). The first difference concerns the construction of the attributes representing the parameters to optimise. Here, we do not need to define a distribution for the variables: we just need boundaries. This is the reason why all the input attributes are `TAttribute` objects, with a minimum and maximum value. The output attributes have no predefined boundaries.

After defining the input and output attributes, we need to create a dataserver to store the results of the optimisation. We provide it only with the input attributes. The output attributes will be automatically added during the optimisation procedure. We also need to tell the cost function which attributes represent the parameters to optimise, and which attributes represents the outputs of the cost function.

The next step is the construction of the optimiser. As explained in Chapter IX, an optimiser needs three objects to perform an optimisation procedure: a dataserver to store the results, a cost function to evaluate the criterion (or the criteria, for multi-objectives optimisation), and an optimisation algorithm to propose a new set of input parameters. The two former objects are already created. In our example, the latter is a `TNloptNelderMead` object.

Once constructed, the optimiser object must receive some additional information before starting the procedure. The first one are the output attributes of the cost function corresponding which correspond to the criteria. In this example, we have only one criterion: the attribute "cost" which refers to the negative log-likelihood of the gaussian process.

The second mandatory information is the starting point of the optimisation. More than one starting point can be provided. As many optimal set of parameters will be stored in the dataserver. Finally, non mandatory options can be set, like the maximum number of evaluations of the cost function, or the tolerance criterion used to consider that the optimisation has converged to a solution. When everything is set, we can launch the optimisation procedure by calling the `solverLoop` function.

Retrieving the optimal parameters is easier in this case than in the LHS: only the optimal solutions for each starting point are stored in the dataserver ! As we have only one starting point in this example, we have only one solution to extract and provide to the `TGPBuilder`. Once done, we can create our kriging model as we did before.

The resulting model is printed as follow:

```
*******************************
** TKriging::printLog[]
*******************************
 Input Variables:      x1:x2:x3:x4
 Output Variable:      y
 Deterministic trend:
 Correlation function: URANIE::Modeler::TMatern32CorrFunction
 Correlation length:   normalised   (not normalised)
                       1.6180e+00 (1.6171e+00 )
                       1.4371e+00 (1.4369e+00 )
                       1.5025e+00 (1.5009e+00 )
                       6.7885e+00 (6.7944e+00 )

 Variance of the gaussian process:      70.8649
 RMSE (by Leave One Out):               0.49911
 Q2:                                    0.849841
*******************************
```

Chapter VI

The Sensitivity module

VI.1 Brief reminder of theoretical aspects

In this section, we will briefly remind the different ways to measure the sensitivity of an output to the inputs of the model. A theoretical introduction is given in [30] to help handle the concept discussed throughout the examples below.

The list of methods available in Uranie will also be briefly discussed, as most of these procedures, local and global ones, are further discussed in the following sections (both implementation and cost in terms of number of assessments).

VI.1.1 Content of the `TSensitivity` class

As for the mathematical function discussed in Section V.3.1.1, the sensitivity classes are now able to cope with constant-size vectors, considering every elements with respect to their counterpart in the other events (leading also to a removal of the previous limitation on the possible number of output attribute to be studied). Most of these classes are based on the `TSensitivity` one, which mainly contains:

- Pointers:
  - to a `TDataServer` object
  - to `TCode` object, and to a `TMethodCall` object (to deal respectively with external code and ROOT-interactive function)
  - to a `TRandom` object (ROOT-class for random drawing)
  - to a `TDSNtupleD` object and to a `TTree` that contains all the results
  - to two `TList` of input and output attributes.
- Methods:
  - **`computeIndexes:`** This is the main method for the evaluation. It checks whether the `TDSNtupleD` is existing (if not, it creates it) and filled with inputs and outputs (if not, it launches either the code or function). There are few possible options common to all the classes that inherit from the `TSensitivity` one:
    - "noIntermediateSaved": This is useful if the code to be launched is quick, to prevent from having every event written to a physical file on disk (it waits the very end of the process to do so, as stated in Section IV.3).
    - "output=": This option reduce the number of output to be considered for the sensitivity analysis. The requested pattern following the equal sign is a usual string where ":" is used as delimiter between two fields. This option is further discussed, considering the constant-size vector case, in the tip box below.

With all this, it calls the `evaluateIndexes` method (providing the same options).

- **`evaluateIndexes:`** Method (empty in `TSensitivity`) in which indices estimation will be done specifically by inheriting-classes. Should not be called by users or with good reasons (discussed with developpers).

- **`fillIndex:`** Never redefined, this method fills the results `TTree` with the method, algorithm, input attribute and value of the sensitivity index (along with its upper and lower 95% confidence interval boundaries).

- **`drawIndexes:`** Draw the measured indices. This method takes three arguments, the first one being the title, the second one a selection cut and the last one the options. The latter parameter is composed of different key words separated by commas. These key words can be:

  * nonewcanv: if not present, a new canvas is made, else it is up to the user to provide one. The difference mostly lies in the memory management and in the way the display is done (if one wants to have multi-pad canvas for instance).
  * pie/hist: whether the resulting plot should be displayed as pie or chart.
  * total/first: to decide which index should be shown (see [30] for more explanations on this aspect).

  As this method is generic for all classes that inherit from the `TSensitivity` one, it will always provide a pie chart with the first indices on the left and another one with the total indices on the right, when the "pie" options is passed, disregarding whether the method under consideration can estimate both kind of index. If it is not the case, the right panel will be a copy of the left one. In this method, the second argument might be in particular use when dealing with more than one output: the `drawIndexes` method assumes that the requested output to be drawn is the first output defined and estimated through your sensitivity analysis. In the case wher several outputs were investigated, the selection field might be used as done below to specified that *tutu* is the output in which one is interested in:

```
tsobol->drawIndexes("What a nice plot","Out==\"tutu\"","all,hist,nonewcanv");
```

Tip All sensitivity classes can deal with vectorial outputs, provided that they are constant-size one. All methods are transparent and it is also possible to restrain the sensitivity analysis to some specific elements of vector. Let's consider y, the output of the code. One can investigate the first, tenth and fifteenth elements by either:

- using the "output=" option (here "output=y[0]:y[10];y[15]"), either passed at the construction or when calling the `computeIndexes` method.
- specifying in a constructor that requested the `const char* out` field, the following list "y[0]:y[10];y[15]", instead of "y".

If both methods are chosen with different elements, the results will be the estimation for all the elements requested.

Tip All sensitivity have been modified for Uranie version greater than 4.1 in order to be able to cope with the Relauncher architecture. A new constructor has been created requesting a pointer to a `TRun` object which has to be chosen amongst the three following types:

- `TSequentialRun`
- `TThreadedRun`
- `TMpiRun`

This object will contain all the information about the way to handle evaluators, whether they are

- C++ function
- Python function
- External Code
- Any composition of instances of previously introduced evaluators

For a more general discussion on these aspects, see Section VIII.2.

With this set, one can look at the different methods listed below.

VI.1.2 List of available methods

Methods for Sensitivity Analysis (SA) are split into two types:

- local: variations around a nominal value,
- global: variations in all the domain.

Currently, one local method and six global methods are implemented for Sensitivity Analysis. All of them have been recently adapted in order to be able to cope with constant-size vector case, considering, as for the statistic, that every element can be considered independent from the other one. This leads to the fact that now, every methods can deal with more than one output as well. The list below provides a concise description of each and every implemented methods. For a more refined theoretical description, please, see [30].

1. **Finite differences** (local method):

   It consists in estimating the partial derivatives around a nominal value for each input parameters (see Section VI.2).

2. **Values Regression** method (linearity):

   It performs a sensitivity analysis based on the coefficients of a normalised linear regression (see Section VI.3).

3. **Ranks Regression** method (monotony):

   Here, the analysis is based on the coefficients of a normalised rank regression (see Section VI.3).

4. **Morris**' screening method:

   It consists in ordering the input variables according to their influence on the output variables. This method should be used for input ranking. Despite the low computational cost encountered, the obtained information is insufficient to get a proper quantitave estimation of the impact of the input variable on the output under consideration (see Section VI.4).

5. **Sobol** method:

   This method produces numerical values for the Sobol indices. However, it requires a high numerical cost as numerous code assessments are needed (see Section VI.5).

   It is based on the so-called *Saltelli & Tarontola* method, to compute the first and total order indices, using different algorithms.

6. **FAST** method:

   It computes Sobol's first order indices from Fourier coefficients, using a sample on a periodic curve with different frequencies for each input variables (see Section VI.6).

7. **RBD** method:

   It computes Sobol's first order indices from Fourier coefficients, using a sample on a periodic curve with an unique frequency (see Section VI.6).

8. **Johnson's relative weights** method:

   It computes the relative weights which are aimed to be a good approximation of the Shapley's values, but whose main advantage is to be a lot quicker to estimate. This method is limited to linear cases (see Section VI.7).

Warning
If the **FAST** or **RBD** methods are to be used, then the *FFTW* library must be installed firstly (and ROOT must have linked installed once FFTW is installed).

All the methods will be applied on the `flowrate` function, introduced in Section IV.1.2.

VI.2 The finite differences method

VI.2.1 General presentation of finite difference sensitivity indices

The finite differences method is among the simplest one. The resulting sensitivity index of an input variable X_i with respect to an output $Y = f(\{X_i\}_{i\in[1,n_X]})$ is an estimation of the derivative of f versus X_i, $\delta f/\delta X_i$, around a nominal value X_i^{nom}. In this implementation of the method, the estimation is obtained by applying an OAT design-of-experiments (*One-At-a-Time*, c.f. Section III.6) to the studied model. For each input's nominal value, we define a range ΔX_i. The resulting estimate of the partial derivative around the nominal value is then given by

$$\frac{\partial f}{\partial X_i} = \frac{f(X_i^{\text{nom}} + \Delta X_i) - f(X_i^{\text{nom}} - \Delta X_i)}{2 \times \Delta X_i}$$

.

VI.2.2 Computation of local sensitivity indices with the finite differences method

In Uranie, computing local sensitivity indices around a nominal value with the finite differences method is dealt with the eponymous class **`TFiniteDifferences`** which inherits from the **`TSensitivity`** class.

This class handles all the steps to compute local indices:

- generating the deterministic sample;
- running the code or the analytic function to get the target attribute;
- computing local indices for each input variable.

VI.2.2.1 Example: simple computation of sensitivity indices with TFiniteDifferences

The example script below uses the **`TFiniteDifferences`** class to compute and display local sensitivity indices:

```
{
 // loading the flowrateModel function
 gROOT->LoadMacro("UserFunctions.C");

 // Define the DataServer  and add the attributes (stochastic variables here)
 TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
 tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
 tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
 tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
 tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
 tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
 tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
 tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
 tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

 // Set nominal values
 tds->getAttribute("rw")->setDefaultValue(0.075);
 tds->getAttribute("r")->setDefaultValue(25000.0);
 tds->getAttribute("tu")->setDefaultValue(90000.0);
 tds->getAttribute("tl")->setDefaultValue(90.0);
 tds->getAttribute("hu")->setDefaultValue(1050.0);
 tds->getAttribute("hl")->setDefaultValue(760.0);
 tds->getAttribute("l")->setDefaultValue(1400.0);
 tds->getAttribute("kw")->setDefaultValue(10500.0);


 // Create a TFiniteDifferences object and compute indexes
 TFiniteDifferences * tfindef = new TFiniteDifferences(tds,"flowrateModel", "rw:r:tu:tl:hu: ↩
    hl:l:kw", "y", "steps=0.5%");
 tfindef->computeIndexes();

 tfindef->getSensitivityMatrix().Print();

  }
```

The result of this script is shown below:

```
1x8 matrix is as follows

     |      0    |      1    |      2    |      3    |      4    |
----------------------------------------------------------------------
   0 |       1019  -3.586e-07   1.265e-09    0.001265      0.1321


     |      5    |      6    |      7    |
----------------------------------------------------------------------
   0 |    -0.1321    -0.02729    0.003639
```

VI.2.2.2 Specifying the input parameters

First, it is necessary to define the uncertain parameters and to add them to a `TDataServer` object:

```
// Define the DataServer
TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
```

```
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

Then, the nominal values of each input parameters must be set (using the method `TAttribute::setDefaultValue`):

```
// Set the nominal values
tds->getAttribute("rw")->setDefaultValue(0.075);
tds->getAttribute("r")->setDefaultValue(25000.0);
tds->getAttribute("tu")->setDefaultValue(90000.0);
tds->getAttribute("tl")->setDefaultValue(90.0);
tds->getAttribute("hu")->setDefaultValue(1050.0);
tds->getAttribute("hl")->setDefaultValue(760.0);
tds->getAttribute("l")->setDefaultValue(1400.0);
tds->getAttribute("kw")->setDefaultValue(10500.0);
```

VI.2.2.3 `TFiniteDifferences` constructor

There are four different constructors to build a **`TFiniteDifferences`** object, each corresponding to a different problem:

- the model is an analytic function run by Uranie,
- the model is a code run by Uranie,
- the outputs of the model are already computed and saved in a `TDataServer` object.
- the model is either a function or a code and the problem is specified through a Relauncher architecture.

VI.2.2.3.1 `TFiniteDifferences` constructor for an analytic function

The constructor prototype used with an analytic function is:

```
// Create a TFiniteDifferences object with an analytic function
TFiniteDifferences(TDataServer *tds, const char *fcn, TString sensitiveAtt, TString ↩
    outputAtt, TString samplingOption="steps=1%")
TFiniteDifferences(TDataServer *tds, void * fcn(double *,double *), TString sensitiveAtt, ↩
    TString outputAtt, TString samplingOption="steps=1%")
```

This constructor takes five arguments:

- a pointer to a `TDataServer` object,
- a pointer to an analytic function (a `const char` that represents the function's name when it has been loaded in ROOT's memory) or a pointer to this function (see Section I.2.5),
- a `TString` to specify the names of the input factors separated by ':' (ex. "rw:r:tu:tl:hu:hl:l:kw"),
- a `TString` to specify the names output variables of the model,
- a `TString` to specify the sample options, its default value is the string "steps=1%".

The options available are the options of the method Sampler::TOATSampling. They are written as strings of the form:"steps=x", where "x" can either be a real value (e.g. "steps=3.1415") or a percentage of the nominal value (e.g. "steps=0.123%"). This "steps" value is the same for each input parameter.

Here is an example of how to use the constructor with an analytic function:

```
TFiniteDifferences * tfindef = new TFiniteDifferences(tds,"flowrateModel", "rw:r:tu:tl:hu: ↩
    hl:l:kw", "y", "steps=0.5%");
```

VI.2.2.3.2 `TFiniteDifferences` constructor for a code

The constructor prototype used with a code is:

```
// Create a TFiniteDifferences object with a code
TFiniteDifferences(TDataServer *tds, TCode *fcode,  TString sensitiveAtt="", TString  ↩
    samplingOption="steps=1%")
```

This constructor takes four arguments:

- a pointer to a `TDataServer` object,
- a pointer to a **`TCode`**,
- a `TString` to specify the names of the input factors separated by ':' (ex. "rw:r:tu:tl:hu:hl:l:kw"), its default value is the empty string ""
- a `TString` to specify the sample options, its default value is the string "steps=1%".

The options available are the options of the method Sampler::TOATSampling. It's a string of the form:"steps=x", where "x" can either be a real value (e.g. "steps=3.1415") or a percentage of the nominal value (e.g. "steps=0.123%"). This "steps" value is the same for each input parameter.

Here is an example of the use of this constructor on the flowrate case:

```
// The reference input file
TString sJDDReference = TString("flowrate_input_with_keys.in");

// Set the reference input file and the key for each input attributes
tds->getAttribute("rw")->setFileKey(sJDDReference, "Rw");
tds->getAttribute("r")->setFileKey(sJDDReference, "R");
tds->getAttribute("tu")->setFileKey(sJDDReference, "Tu");
tds->getAttribute("tl")->setFileKey(sJDDReference, "Tl");
tds->getAttribute("hu")->setFileKey(sJDDReference, "Hu");
tds->getAttribute("hl")->setFileKey(sJDDReference, "Hl");
tds->getAttribute("l")->setFileKey(sJDDReference, "L");
tds->getAttribute("kw")->setFileKey(sJDDReference, "Kw");

// The output file of the code
TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
// The attribute in the output file
fout->addAttribute(new TAttribute("yhat"));

// Instanciation de mon code
TCode *mycode = new TCode(tds, "flowrate -s -k");
// Adding the outputfile to the tcode object
mycode->addOutputFile( fout );

TFiniteDifferences * tfindefC = new TFiniteDifferences(tds,mycode);
```

VI.2.2.3.3 **TFiniteDifferences constructor for a runner**

The constructor prototype used with a code is:

```
// Create a TFiniteDifferences object with a runner
TFiniteDifferences(TDataServer *tds, TRun *run,  TString sensitiveAtt="", TString  ↩
    samplingOption="steps=1%")
```

This constructor takes four arguments:

- a pointer to a `TDataServer` object,
- a pointer to a **`TRun`**,
- a `TString` to specify the names of the input factors separated by ':' (ex. "rw:r:tu:tl:hu:hl:l:kw"), its default value is the empty string ""
- a `TString` to specify the sample options, its default value is the string "steps=1%".

The options available are the options of the method Sampler::TOATSampling. It's a string of the form:"steps=x", where "x" can either be a real value (e.g. "steps=3.1415") or a percentage of the nominal value (e.g. "steps=0.123%"). This "steps" value is the same for each input parameter.

Here is an example of the use of this constructor on the flowrate case, using the code:

```
// The input file
TKeyScript infile("flowrate_input_with_keys.in");
// provide the input and their key
infile.setInputs(8, tds->getAttribute("rw"), "Rw", tds->getAttribute("r"), "R",
tds->getAttribute("tu"), "Tu", tds->getAttribute("tl"), "Tl", tds->getAttribute("hu"), "Hu" ↩
    ,
tds->getAttribute("hl"), "Hl", tds->getAttribute("l"), "L", tds->getAttribute("kw"), "Kw");

TAttribute yhat("yhat");
// The output file of the code
TKeyResult outfile("_output_flowrate_withKey_.dat");
// The attribute in the output file
outfile.addOutput(&yhat, "yhat");

// Instanciation de mon code
TCodeEval code("flowrate -s -k");
// Adding the intput/output file to the code
code.addInputFile(&infile);
code.addOutputFile(&outfile);

TSequentialRun run(&code);
run.startSlave();
if(run.onMaster())
{
   TFiniteDifferences * tfindefR = new TFiniteDifferences(tds, &run);
   // ....
}
```

VI.2.2.3.4 **TFiniteDifferences constructor using a filled TDS**

The two constructors before are used for cases where the computation of the model outputs are run from Uranie. However it is possible to compute the outputs of the model outside of Uranie then load them in a `TDataServer` object (via a file) and use that `TDataServer` object to compute the finite difference indices.

Be aware that in that case, because of the use of internal variables, it is necessary to use a sample of input parameters computed with Uranie, exported in a file and then completed with the outputs of the model.

The constructor prototype used with a `TDataServer` object already containing the simulations is:

```
 //  Create a TFiniteDifferences object with already filled TDS
TFiniteDifferences(TDataServer *tds,  TString inputAtt, TString outputAtt, TString ↩
    sensitiveAtt=TString(""))
```

This constructor takes four arguments:

- a pointer to a `TDataServer` object filled with data,
- a `TString` to specify the names of all the input factors separated by ':' (ex. "rw:r:tu:tl:hu:hl:l:kw"),
- a `TString` to specify the names of the output variables of the model,
- a `TString` to specify the names of the inputs of the model which are studied, its default value is the empty string "".

Below is an example of the constructor with a `TDataServer` object filled:

```
// Define the DataServer  and add the attributes
TDataServer *tdsFilled = new TDataServer();
tdsFilled->fileDataRead("sampleFiniteDifferencesFlowrateModel.dat");
tdsFilled->getAttribute("flowrateModel")->setOutput(); // Mark the output

TFiniteDifferences * tfindef2 = new TFiniteDifferences(tdsFilled, TString("rw:r:tu:tl:hu:hl ↩
    :l:kw"), TString("flowrateModel"));
```

Warning
This constructor uses a `TDataServer` object already filled with specific internal variables (__nominal_set__ and __modified_att__) and a specific sample!
There are several conditions to use it:

- use the constructor without argument for the `TDataServer`;
- the input factors sample must have been generated with the method `TFiniteDifferences::generateSample`;
- the output factors of the model must have been specified with the method `TAttribute::setOutput`;

VI.2.2.4 Computing the indices

To compute the indices, run the method `computeIndexes`:

```
tfindef->computeIndexes();
```

Note that this method is all inclusive: it constructs the sample (if it does not exist), launches the simulations (if they are not already computed) and computes the indices.

VI.2.2.5 Extracting the indices

To extract the indices use the method `getSensitivityMatrix`:

```
TMatrixD matRes = tfindef->getSensitivityMatrix();
matRes.Print();
```

Summary: `TFiniteDifferences`

- `TFiniteDifferences`(**`TDataServer`** *tdsNominal, **void/const char** *fcn, **`TString`** sensitiveAtt, **`TString`** outputAtt, **`TString`** samplingOption="steps=1%")
  - **tdsNominal**: an empty dataserver containing the input variables. Each input variables must have a default value set used as the nominal value.
  - **fcn**: the function to analyse (either a `void` or a `const char`).
  - **sensitiveAtt**: A list of attribute names separated by colons that will be taken into account for the SA.
  - **outputAtt**: attribute names which represent the responses of the problem.
  - **samplingOption**: The options for the OAT sampling (Default = "steps=1%").
- `TFiniteDifferences`(**`TDataServer`** *tdsNominal, **`TCode`** *fcode, **`TString`** sensitiveAtt="", **`TString`** samplingOption="steps=1%")
  - **tdsNominal**: an empty dataserver containing the input variables. Each input variables must have a default value set, used as the nominal value.
  - **fcode**: The code to analyse.
  - **sensitiveAtt**: input attribute names separated by colons that will be taken into account for the SA. If the string is equal to "" or "*", all the attributes are taken into account.
  - **samplingOption**: The options for the OAT sampling (Default = "steps=1%").
- `TFiniteDifferences`(**`TDataServer`** *tdsNominal, **`TRun`** *run, **`TString`** sensitiveAtt="", **`TString`** samplingOption="steps=1%")
  - **tdsNominal**: an empty dataserver containing the input variables. Each input variables must have a default value set used as the nominal value.
  - **run**: The runner to be used that contains the code and the way to distribute.
  - **sensitiveAtt**: input attribute names separated by colons taken into account for the SA. If the string is equal to "" or "*", all the attributes are taken into account.
  - **samplingOption**: The options for the OAT sampling (Default = "steps=1%").
- `TFiniteDifferences`(**`TDataServer`** *tdsNominal, **`TString`** inputAtt, **`TString`** outputAtt, **`TString`** sensitiveAtt=TString(""))
  - **tdsNominal**: this `TDataServer` object contains the input and output values necessary for the finite differences calculation. It must also contain the attributes "__nominal_set__" and "__modified_att__" which are necessary to know which attribute is modified.
  - **inputAtt**: input attribute names separated by colons. They must refer to existing attributes in the "tdsNominal".
  - **outputAtt**: attribute name which represent the unique response of the problem.
  - **sensitiveAtt**: A list of attribute names separated by colons taken into account for the SA. If the string is equal to "" or "*", all the input attributes are taken into account.
- void `computeIndexes`(**Option_t** *option=""): Compute sensitivity indices.
  - **option**: no option has been implemented.
- `TMatrixD getSensitivityMatrix`(): Get the matrix containing the indices.

VI.3 The regression method

VI.3.1 General presentation of regression's coefficients

The `TRegression` class is dealing with computing the SRC and SRRC coefficients but also the PCC and PRCC ones (for a definition of these coefficients, see [30]). The choice of which values to be computed between these four is set at the construction by precising respectively "src", "srrc", "pcc" or "prcc" and is no more inclusive. It is indeed now possible to get, for example, the results of both SRC and PRCC estimation by passing the option "srcprcc", **for one or more outputs**. This class computes the regression coefficients from the observations of the model contained in a `TDataServer`. The construction of the data sample is the user burden and the quality of the regression coefficients computed will depend upon it (unless analysing a dataset out of an experiment).

In the case where SRC or SRRC coefficients are requested, a second estimation is performed, based on correlation coefficients (between the output and the input under consideration, see [30] for completness). This estimation is costless and allows to get an idea of the 95% confidence-interval (CI) of the estimated coefficient. This CI has to be considered as a very good guess of the 95% CI and could only be considered exact, if one respects the underlying hypothesis for its estimation: all input variables should be gaussian-distributed variables. An illustration of the way to look at this CI is shown in Section VI.3.2.5.1.

Using the `TRegression` class is simple, as there is only one possible creator, and the only other method to be called is `computeIndexes`. The rest is common to any of the following SA classes.

VI.3.2 Computation of the coefficients with Uranie

Computing Standardised Regression Coefficients (**SRC**), Standardised Rank-Regression Coefficients (**SRRC**), partial correlation coefficient (**PCC**) and partial rank correlation coefficient (**PRCC**) in Uranie is dealt with the **`TRegression`** class which inherits from the **`TSensitivity`** class. The following sections will explain how to compute and handle the requested coefficients.

VI.3.2.1 Example: simple computation of SRC and SRRC coefficients

The example script uses the **`TRegression`** class to compute and display the **SRC** and **SRRC** coefficients:

```
{
 // Define the DataServer and fill it with datas in a file
 TDataServer *tds = new TDataServer("tdsflowrate", "DataBase flowrate");
 tds->fileDataRead("sampleLHSFlowrateModel4000points.dat");

 // Create a TRegression object and compute indexes
 TRegression * treg = new TRegression(tds, "rw:r:tu:tl:hu:hl:l:kw","flowrateModel", " ↩
     SRCSRRC");
 treg->computeIndexes();

 // Draw SRC Indexes
 TCanvas *cc = new TCanvas("canpie-SRC", "Pie chart SRC",10,32,1200,800);
 TPad *apad = new TPad("apad","apad",0, 0.03, 1, 1);  apad->Draw();  apad->cd();
 treg->drawIndexes("Flowrate", "", "nonewcanv,pie,SRC");

 // Draw SRRC Indexes
 TCanvas *ccc = new TCanvas("canpie-SRRC", "Pie chart SRRC",10,32,1200,800);
 TPad *pad2 = new TPad("pad2","pad2",0, 0.03, 1, 1); pad2->Draw(); pad2->cd();
 treg->drawIndexes("Flowrate", "", "nonewcanv,pie,SRRC");

  }
```

In this script, the observations are data loaded from a file into a `TDataServer` object, a **`TRegression`** object is created to compute the coefficients. Then both indices are computed by using the method `TSensitivity::computeIndex`
Finally, SRC and SRRC coefficients are graphically displayed in pie charts, respectively in Figure VI.1 and Figure VI.2, with the `TSensitivity::drawIndexes` method.

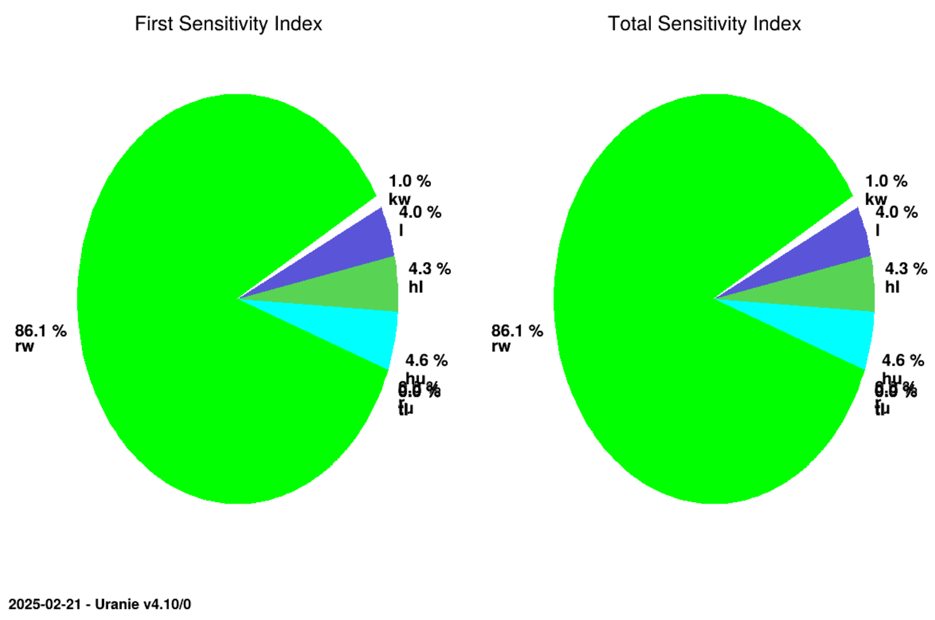


Figure VI.1: SRC coefficients estimated for the `flowrate` function.

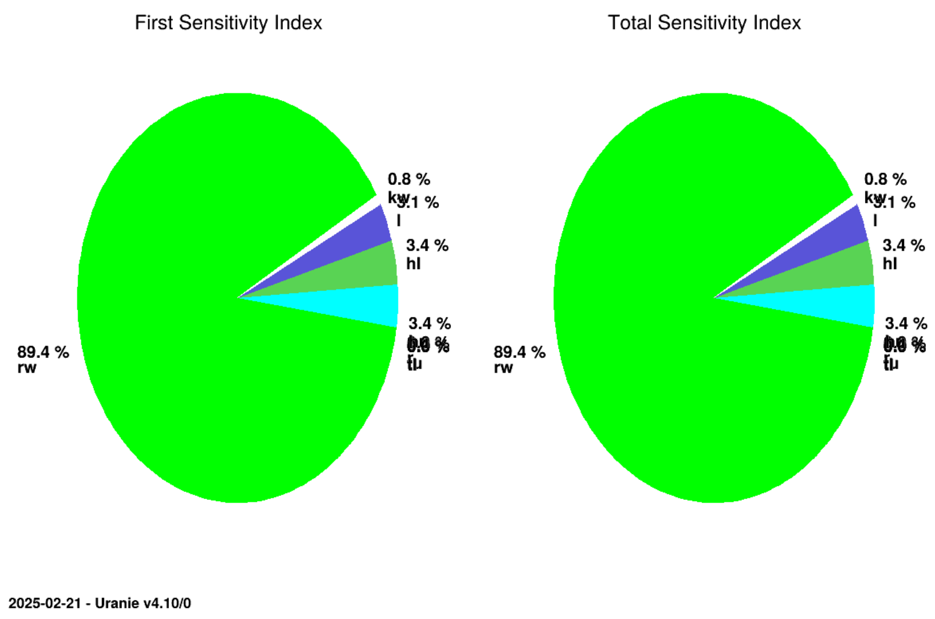


Figure VI.2: SRRC coefficients estimated for the `flowrate` function.

VI.3.2.2 `TRegression` constructor

To build a **`TRegression`** object, use the following constructor:

```
TRegression(TDataServer *tds, const char* varinput, const char* varoutput, Option_t  * ↩
    option="src")
```

The **`TRegression`** constructor needs:

- a pointer to a `TDataServer` object where the input and output attributes are stored,
- a string containing the input parameter names separated by colons (ex. "rw:r:tu:tl:hu:hl:l:kw"),
- a string containing the name of the model's outputs (ex. "flowrateModel"),
- a string containing the type of coefficient to compute ("SRC", "SRRC", "PCC", or "PRCC"), its default value is the string "SRC".

The creation of a **`TRegression`** object for computing the **`SRC`** and the **`SRRC`** coefficients is therefore:

```
// Create a TRegression object and compute SRC indices
TRegression * treg = new TRegression(tds, "rw:r:tu:tl:hu:hl:l:kw","flowrateModel", "SRCSRRC ↩
    ");
```

VI.3.2.3 Computing the SRC and SRRC indices

The computation of the coefficients (also call indices) is done with the method `computeIndexes`:

```
void TSensitivity::computeIndexes(Option_t * option)
```

In our example, its use is therefore:

```
 // Compute the indices
treg->computeIndexes();
```

VI.3.2.4 Displaying the indices

To display graphically the coefficients, use the `drawIndexes` method:

```
void TSensitivity::drawIndexes(TString sTitre, const char *select, Option_t * option)
```

The method needs:

- a `TString` containing the title of the figure,
- a string containing a selection (empty if no selection),
- a string containing the options of the graphics separated by commas.

Some of the options available are:

- "nonewcanv": to not create a new canvas,
- "pie": to display a pie chart,
- "hist": to display a histogram,

- "SRC": to display the SRC indices,
- "SRRC": to display the SRRC indices,
- "PCC": to display the PCC indices,
- "PRCC": to display the PRCC indices,

In our example the use of this method is:

```
// Draw SRC Indices
TCanvas *cc = new TCanvas("canpie-SRC", "Pie chart SRC");
treg->drawIndexes("Flowrate", "", "nonewcanv,pie,SRC");

// Draw SRRC Indices
TCanvas *ccc = new TCanvas("canpie-SRRC", "Pie chart SRRC");
treg->drawIndexes("Flowrate", "", "nonewcanv,pie,SRRC");
```

Here is another example with a histogram figure:

```
TCanvas *cccc = new TCanvas("canhist-SRC", "Hist chart SRC");
treg->drawIndexes("Flowrate", "", "nonewcanv,hist,first");
```

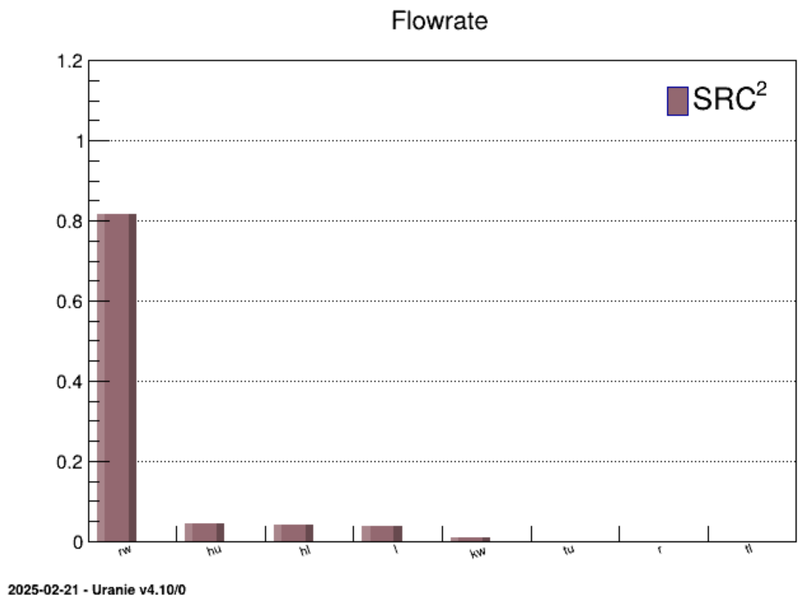


Figure VI.3: Histogram of SRC coefficients

VI.3.2.5 Extracting the coefficients

The coefficients, once computed, are stored in a `TTree`. To get this `TTree`, use the method `TSensitivity::getResul`

```
TTree * results = treg->getResultTuple();
```

Several methods exist in ROOT to extract data from a `TTree`, it is advised to look for them into the ROOT documentation. We propose two ways of extracting the value of each coefficient from the `TTree`.

VI.3.2.5.1 First method of extraction

The first method use the method `getValue` of the `TRegression` object specifying the order of the extract value, the related input and possibly more selected options.

```
treg->getValue("First","hl");
```

VI.3.2.5.2 Second method of extraction

The second method uses 3 steps to extract an index:

- scan the `TTree` for the chosen input variable (with a selection) in order to obtain its row number. In our example, if we chose the variable "hl", we'll use the command:

  ```
  results->Scan("*","((Inp==\"hl\")&&(Method==\"SRC^2\"))");
  ```

  This results in the following table, in which the SRC coefficient of "hl" is in the row 40:

  ```
  ************************************************************************************
  *    Row   *   Out *  Inp * Order * Method *       Algo *   Value * CILower * CIUpper *
  ************************************************************************************
  *       40 * flowr *   hl * First *   SRC^2 * --first-- * 0.04102 *       -1 *       -1 *
  *       41 * flowr *   hl * Total *   SRC^2 * --total-- * 0.04102 *       -1 *       -1 *
  *       42 * flowr *   hl * First *   SRC^2 * --rho^2-- * 0.04124 * 0.03025 * 0.05353 *
  *       43 * flowr *   hl * Total *   SRC^2 * --rho^2-- * 0.04124 * 0.03025 * 0.05353 *
  ************************************************************************************
  ```

  The numbers stored in row 42 are the estimation using the correlation coefficient between the output and the input under consideration, and they are shown to crosscheck the main estimation and to display the estimated 95% confidence interval (CI), see Section VI.3.1 and [30] for more explanations.

- set the entry of the `TTree` on this row with the method `GetEntry`;

- get the value of the index with `GetValue` method on the "Value" leaf of the `TTree`.

Below is an example of extraction of the index for the variable "hl" in our flowrate case:

```
results->Scan("*","((Inp==\"hl\")&&(Method==\"SRC^2\"))");
results->GetEntry(20);
Double_t Rw_Indexe =  results->GetLeaf("Value")->GetValue();
```

VI.3.2.5.3 Third method of extraction

The second method uses 2 steps to extract an index:

- use the `Draw` method with a selection to select the index, for example the selection for the SRC coefficient of "rw" is "Inp==\"rw\" && Algo==\"--first--\"";

- get the pointer on the value of the index with the `GetV1` method on the `TTree`.

Below is another example of extraction of the index for the variable "rw" in our flowrate case:

```
results->Draw("Value","Inp==\"rw\" && Algo==\"--first--\" && Method==\"SRC^2\" ","goff");
Rw_Indexe = results->GetV1()[0];
```

VI.3.2.6 Getting R^2 and R^2_{adj}

To evaluate the pertinence of the indices, it is recommended to check the value of both R^2 and R^2_{adj}. The use of the methods `TRegression::getR2` and `TRegression::getR2A` is now deprecated: as one can work both on rank and values at the same time, the quality criteria can be computed for all the estimations and so they have to be kept in the ntuple result as well. The numerical values can then be retrieved as previously explained, an example is shown below to get the one from the regression on values (SRC).

Example:

```
// coefficient of determination R2
results->Draw("Value","Inp==\"__R2__\" && Algo==\"--first--\" && Method==\"SRC^2\" ","goff" ↩
    );
Rw_Indexe = results->GetV1()[0];

// adjusted coefficient of determination R2A
results->Draw("Value","Inp==\"__R2A__\" && Algo==\"--first--\" && Method==\"SRC^2\" ","goff ↩
    ");
Rw_Indexe = results->GetV1()[0];
```

VI.3.2.7 Getting the sum of squared indices

As stated in [30], it can be interesting to consider the sum of the squared indices (in particular for SRC coefficients). As for the quality criteria, discussed previously, this computation can be done for all the estimations and so it has to be kept in the ntuple result. The numerical values can then be retrieved as previously explained, an example is shown below to get the ones from the regression on values (SRC).

Example:

```
// coefficient of determination sum
results->Draw("Value","Inp==\"__sum__\" && Algo==\"--first--\" && Method==\"SRC^2\" ","goff ↩
    ");
Rw_Indexe = results->GetV1()[0];
```

Summary: `TRegression`

- `TRegression`(**`TDataServer`** *tds, **const char*** varinput, **const char*** varoutput, **Option_t** *option="src")
  - **tds**: a dataserver containing the attributes of the input and output variables and their associated data.
  - **varinput**: input attribute names separated by colons. They represent all the inputs of the problem. They must refer to existing attributes in the "tds".
  - **varoutput**: attribute names which represent the responses of the problem. it must refer to existing attributes in the "tds".
  - **option**: a string containing the type of regression coefficients to compute: "SRC", "SRRC"; "PCC" or "PRCC" (Default = "src").
- void `computeIndexes`(**Option_t** *option=""): Compute regression coefficients.
  - **option**: no option has been implemented.
- void `drawIndexes`(**`TString`** sTitre, **const char*** select="", **Option_t** *option=""): draw the coefficients.
  - **sTitre**: a string containing the title of the figure.
  - **select**: a string containing a solution.
  - **option**: a string containing the options of the graphics separated by commas.
- `TTree` * `getResultTuple`(): Get the tree containing the indices.

VI.4 The Morris screening method

VI.4.1 Principle of the Morris' method

The Morris method is an effective screening procedure that robustifies a bit the OAT protocol (*One-factor-At-a-Time*, see Section III.6.1). Instead of varying every input parameters only once (leading then to a minimum of $n_X + 1$ assessments of the code/function, with an OAT technique), the Morris method repeats this OAT principle r times. More precision on this method can be found in [30]. The resulting cost (in terms of assessment number) is then $r(n_X + 1)$.

The results are usually visualised in the (μ^*,σ) plane, allowing to sort its inputs in the following categories:

- factors that have negligible effects on the output: both μ^* and σ are small.
- factors that have linear effects, without interaction with other inputs: μ^* is large (all variations have an impact) but σ is small (the impact is the same independently of the starting point).
- factors that have non-linear effects and/or interaction with other inputs: both μ^* and σ are large.

In terms of implementation, there are only few methods that can be called once the `TMorris` object is constructed (constructed either with a code, a function or an already-filled `TDataServer`):

- `generateSample:` it produces the different patterns of OAT
- `computeIndexes:` it computes the indices and stores them in the result `TTree`.
- `drawIndexes:` the results are presented in the (μ,σ) plane, unless the option "mustar" is used to get μ^* as X axis instead.

Warning
If the `TMorris` object is constructed to run a function, the output used for the estimation will be the first output provided by the function, unless the sixth argument of the constructor is filled.

VI.4.2 The Morris' method in Uranie

Computing Morris coefficients μ, μ^* and σ in Uranie is dealt with the **`TMorris`** class which inherits from the **`TSensitivity`** class.

Warning
The **Morris** method has been designed initially to deal with uniform probability law and uniform only. The implementation has been extended in Uranie by allowing all probability laws, but this has to be done with caution: if one is considering to use infinite-based law it is crucial to set bounds to these. The probability space is divided in a $[0,1]^{nx}$ hyper-grid with p level on each dimension, and by definition, 0 and 1 are possible values that are very likely to arise. When going back from probability space to physical one, their corresponding response should be respectively $-\infty$ and $+\infty$, this is why all infinite-based law should be bounded. As the chosen bounds will be very likely to be used, their values should be chosen with care.
From version 4.4.0, the `TMorris` will quit automatically if an at least one infinite-based law is unbounded static that:

```
<URANIE::ERROR>
<URANIE::ERROR> *** URANIE ERROR ***
<URANIE::ERROR> *** File[/[...]/souRCE/meTIER/sensitivity/souRCE/TMorris.cxx] Line ↩
    [244]
<URANIE::ERROR>  Morris method is not meant to be run with unbounded infinite  ↩
    stochastic laws [kNus].
<URANIE::ERROR>  The sampling can indeed, AND WILL CERTAINLY, randomly draw a  ↩
    probability of 0 or 1 in the [0,1]^{nX} hypercube (see methodology).
<URANIE::ERROR>  The theoretical answer, given an unbounded infinite stochastic laws ↩
    , is repectively - infinite or + infinite.
<URANIE::ERROR>  Please, if you want to use infinite stochastic laws, use the  ↩
    setBounds(lower,upper) method to truncate the law.
<URANIE::ERROR>  This would allow you to define the proper physical space in which  ↩
    your problem is defined.
<URANIE::ERROR>  !!!! WARNING !!!:
<URANIE::ERROR>   Bear in mind that the bound values will very likely be used, so  ↩
    choose them wisely (for a gaussian, few sigma away from the mean but not much).
<URANIE::ERROR>  !!!! WARNING !!!.
<URANIE::ERROR> *** END of URANIE ERROR ***
<URANIE::ERROR>
```

This class handles all the steps to compute the Morris coefficients:

- generating the sample of Morris trajectories;
- running the code or the analytic function to get the response on the sample;
- computing the values of μ, μ^* and σ for each input variable.

VI.4.2.1 Example: simple computation of computing the sensitivity screening indices

The example script below uses the **`TMorris`** class to compute and display Morris sensitivity screening measures:

```
{
 gROOT->LoadMacro("UserFunctions.C");

 // Define the DataServer
 TDataServer *tds = new TDataServer("tdsflowrate", "DataBase flowrate");
 tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
 tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
 tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
 tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
 tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
 tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
 tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
 tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

 // Set the Morris method parameters
 Int_t nreplica = 3;
 Int_t nlevel = 10;

 // Create a TMorris object
 TMorris * scmo = new TMorris(tds, "flowrateModel", nreplica, nlevel);

 // generate a Morris sample
 scmo->generateSample();

 //Save the sample
 tds->exportData("_morris_sampling_.dat");

 // compute the Morris screening measures
 scmo->computeIndexes();

 // Display graphically the Morris screanig measures
 TCanvas *cc = new TCanvas("can", "histogramme");
 TPad *apad = new TPad("apad","apad",0, 0.03, 1, 1); apad->Draw(); apad->cd();
 scmo->drawIndexes("mustar,nonewcanv");

  }
```

Below is the figure generated by this script:

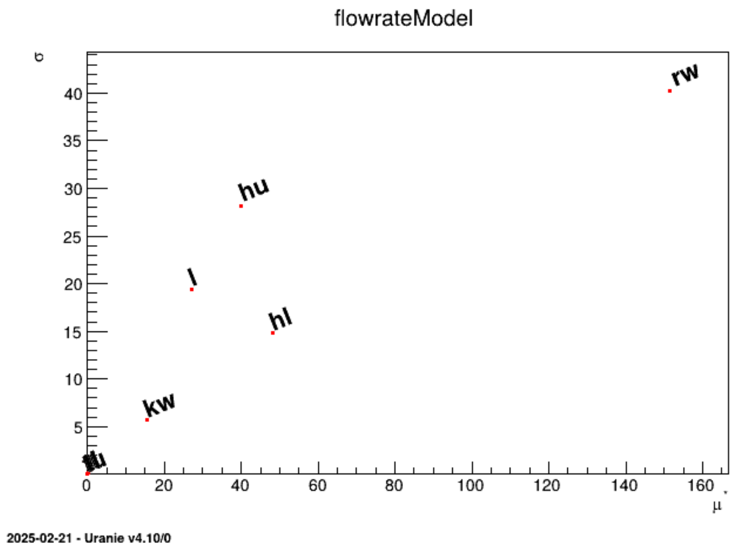


Figure VI.4: Morris screening indices

VI.4.2.2 Specifying the input parameters

First, one defines the uncertain parameters and adds them to a `TDataServer` object:

```
// Define the DataServer
TDataServer *tds = new TDataServer("tdsflowrate", "DataBase flowrate");
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

VI.4.2.3 **`TMorris`** constructors

There are four different constructors to build a **`TMorris`** object, each corresponding to a different problem:

- the model is an analytic function run by Uranie,
- the model is a code run by Uranie,
- the outputs of the model are already computed and saved in a `TDataServer` object.
- the model is either a function or a code and the problem is specified through a Relauncher architecture.

VI.4.2.3.1 **`TMorris`** constructor for an analytic function

The constructor prototype used with an analytic function is:

```
// Create a TMorris object with an analytic function
TMorris(TDataServer *tds,const char *fcn,Int_t nreplica, Int_t level, Double_t delta=0, ↩
    TString sout="")
TMorris(TDataServer *tds,void *fcn,TString sinp,TString sout,Int_t nreplica, Int_t level, ↩
    Double_t delta=0)
```

This constructor takes six arguments:

- a pointer to a `TDataServer` object,
- a pointer to an analytic function (either a `void` or a `const char` that represents the function's name when it has been loaded in ROOT's memory),
- an integer to specify the number of trajectories the object will generate,
- an integer to specify the level of the grid which will be used
- an integer to specify the delta parameter, with $\Delta \in [1, level-1]$,
- a `TString` to specify the names of the input variables of the model.
- a `TString` to specify the names of the output variables of the model.

Here is an example of how to use the constructor with an analytic function:

```
Int_t nreplica = 3;
Int_t nlevel = 10;
TMorris * scmo = new TMorris(tds, "flowrateModel", nreplica, nlevel);
```

VI.4.2.3.2 `TMorris` constructor for a code

The constructor prototype used with a code is:

```
// Create a TMorris object with a code
TMorris(TDataServer *tds,TCode *fcode, Int_t nreplica, Int_t level,  Double_t delta=0)
```

This constructor takes five arguments:

- a pointer to a `TDataServer` object,
- a pointer to a **`TCode`**,
- an integer to specify the number of trajectories the object will generate,
- an integer to specify the level of the grid which will be used
- an integer to specify the delta parameter, with $\Delta \in [1, level-1]$,

Here is an example using this constructor for the flowrate case:

VI.4.2.3.3 **`TMorris` constructor for a runner**

The constructor prototype used with a runner object is:

```
// Create a TMorris object with a runner
TMorris(TDataServer *tds,TRun *run, Int_t nreplica, Int_t level,  Double_t delta=0)
```

This constructor takes five arguments:

- a pointer to a `TDataServer` object,
- a pointer to a **`TRun`**,
- an integer to specify the number of trajectories the object will generate,
- an integer to specify the level of the grid which will be used
- an integer to specify the delta parameter, with $\Delta \in [1, level - 1]$,

Here is an example using this constructor for the flowrate code in sequential mode:

```
// The input file
TKeyScript infile("flowrate_input_with_keys.in");
// provide the input and their key
infile.setInputs(8, tds->getAttribute("rw"), "Rw", tds->getAttribute("r"), "R",
tds->getAttribute("tu"), "Tu", tds->getAttribute("tl"), "Tl", tds->getAttribute("hu"), "Hu" ↩
    ,
tds->getAttribute("hl"), "Hl", tds->getAttribute("l"), "L", tds->getAttribute("kw"), "Kw");

TAttribute yhat("yhat");
// The output file of the code
TKeyResult outfile("_output_flowrate_withKey_.dat");
// The attribute in the output file
outfile.addOutput(&yhat, "yhat");

// Instanciation de mon code
TCodeEval code("flowrate -s -k");
// Adding the intput/output file to the code
code.addInputFile(&infile);
code.addOutputFile(&outfile);

TSequentialRun run(&code);
run.startSlave();
if(run.onMaster())
{
    Int_t nreplicas = 3;
    Int_t nlevels = 10;
    TMorris * scmoR = new TMorris(tds, &run, nreplicas, nlevels);
    // ...
}
```

VI.4.2.3.4 **`TMorris` constructor using a filled `TDataServer` object**

The two constructors before are used in the cases where the computation of the model outputs are run from Uranie. However it is possible to compute the outputs of the model outside of Uranie then load them in a `TDataServer` object (via a file) and use that `TDataServer` object to compute the finite differences indices.

Warning

This constructor uses a `TDataServer` object already filled with specific internal variables (__npoints__, __directions__ and __morris__difference__) and a specific sample!

There are several conditions to use it:

- use the constructor without argument for the `TDataServer`;
- the input factors sample must have been generated with the method `TMorris::generateSample`.

The constructor prototype used with a `TDataServer` object already containing the simulations is:

```
//  Create a TMorris object with already filled TDS
TMorris(TDataServer *tds,  const char *inp,  const char *out,  Option_t *option="")
```

This constructor takes three arguments:

- a pointer to a `TDataServer` object filled,
- a string to specify the names of the input factors separated by ':' (ex. "rw:r:tu:tl:hu:hl:l:kw"),
- a string to specify the names of the output variables of the model.

Below is an example of the constructor with a `TDataServer` object filled:

```
// Define the DataServer
TDataServer *tds = new TDataServer();
tds->fileDataRead("_morris_launching_.dat");

TMorris * scmo = new TMorris(tds, "rw:r:tu:tl:hu:hl:l:kw", "flowrateModel");
```

VI.4.2.4 Generating the sample

To generate the Morris sample, use the `generateSample` method:

```
scmo->generateSample();
```

Then, the sample generated can be exported in a file and used outside of Uranie to compute the simulations associated.

VI.4.2.5 Computing the indices

To compute the sensitivity screening measures, use the method `computeIndexes`:

```
scmo->computeIndexes();
```

Note that this method is all inclusive: it constructs the sample (if it does not exist), launches the simulations (if they are not already computed) and computes the indices.

VI.4.2.6 Displaying the indices

To display graphically the coefficients, use the method `drawIndexes`.

This method shows by default the value of the measurements of σ versus the measurements of μ for all input factors of the model. To display σ versus μ^* the argument "mustar" should be given, as follow:

```
TCanvas *cc = new TCanvas("can", "histgramme");
scmo->drawIndexes("mustar");
```

VI.4.2.7 Extracting the measures

The coefficients, once computed, are stored in a `TTree`. To get this `TTree` use either the `TSensitivity::getResult` or the `TMorris::getResultTuple()` that will provide the dedicated Morris ntuple.

```
TTree * ntresu = scmo->getMorrisResults();
```

Several methods exist in ROOT to extract data from a `TTree`, it is advised to look for them into the ROOT documentation. We propose two ways of extracting the value of each coefficient from the `TTree`.

VI.4.2.7.1 First method of extraction

The first method use the method `getValue` of the `TMorris` object specifying the order of the extract value ("mustar", "mu" ou "sigma"), the related input and possibly more selected options.

```
double hl_mustar_index = scmo->getValue("mustar","hl");
double hl_mu_index = scmo->getValue("mu","hl");
double hl_sigma_index = scmo->getValue("sigma","hl");
```

VI.4.2.7.2 Second method of extraction

The second method uses 3 steps to extract an index:

- scan the `TTree` for the chosen input variable (with a selection) in order to obtain its row number. In our example, if we chose the variable "hl", we'll use the command:

  ```
  ntresu->Scan("*","Input==\"hl\"");
  ```

  and in the resulting figure below we see that the measure μ of "hl" is in the row 5:

  ```
  ************************************************************************
  *    Row   *      Input *     Output *      mu.mu * mustar.mu * sigma.sig *
  ************************************************************************
  *        5 *         hl * flowrateM * -48.15217 * 48.152173 * 14.855638 *
  ************************************************************************
  ```

- set the entry of the `TTree` on this row with the method `GetEntry`;
- get the value of the index with `GetValue` method on the "mu" leaf of the `TTree`.

Below is an example of extraction of the index μ for hl in our flowrate case:

```
ntresu->Scan("*","Input==\"hl\"");
ntresu->GetEntry(5);
Double_t Hl_mu_Indexe =  ntresu->GetLeaf("mu")->GetValue();
```

VI.4.2.7.3 Third method of extraction

The third method uses 2 steps to extract an index:

- use the `Draw` method with a selection to select the index, for example the selection for the measure μ^* of "rw" is "Input==\"rw\"";

- get the pointer on the value of the index with `GetV1` method on the `TTree`.

Below is another example of extraction of the index μ^* for "rw" in our flowrate case:

```
ntresu->Draw("mustar","Input==\"rw\"","goff");
Double_t Rw_mustar_Indexe = ntresu->GetV1()[0];
```

Summary: `TMorris` constructor

- `TMorris`(**`TDataServer`** *tds, **const char** *fcn, **Int_t** nreplica, **Int_t** level, **Double_t** delta=0, **`TString`** sout="")
  - **tds**: an empty dataserver containing the input variables.
  - **fcn**: the function to analyse (the function's name).
  - **nreplica**: an integer specifying the number of trajectories which will be generated.
  - **level**: an integer specifying the number of intervals in each dimension for the grid.
  - **delta**: an integer value to specify the delta parameter, with $\Delta \in [1, level - 1.$
  - **sout**: attribute names which represent the responses of the model.
- `TMorris`(**`TDataServer`** *tds, **void** *f(*double *,double **), **`TString`** sin, **`TString`** sout, **Int_t** nreplica, **Int_t** level, **Double_t** delta=0, **`TString`** sout="")
  - **tds**: an empty dataserver containing the input variables.
  - **f**: the function to analyse (a pointer).
  - **sin**: attribute names which represent the inputs of the model.
  - **sout**: attribute names which represent the responses of the model.
  - **nreplica**: an integer specifying the number of trajectories which will be generated.
  - **level**: an integer specifying the number of intervals in each dimension for the grid.
  - **delta**: an integer value to specify the delta parameter, with $\Delta \in [1, level - 1.$
- `TMorris`(**`TDataServer`** *tds, **const char** *inp, **const char** *out, **Option_t** *option="")
  - **tds**: a dataserver containing the input and output attributes and their associated data.
  - **inp**: input attribute names separated by colons. They represent all the inputs of the problem. They must refer to existing attributes in the "tds".
  - **out**: attribute names which represent the response of the problem. it must refer to existing attributes in the "tds".
  - **option**: no option has been implemented.
- `TMorris`(**`TDataServer`** *tds, **`TCode`** *fcode, **Int_t** nreplica, **Int_t** level, **Double_t** delta=0)
  - **tds**: an empty dataserver containing the input variables.
  - **fcode**: the code to analyse .
  - **nreplica**: an integer specifying the number of trajectories which will be generated.
  - **level**: an integer specifying the number of intervals in each dimension for the grid.
  - **delta**: an integer value to specify the delta parameter, with $\Delta \in [1, level - 1.$
- `TMorris`(**`TDataServer`** *tds, **`TRun`** *run **Int_t** nreplica, **Int_t** level, **Double_t** delta=0)
  - **tds**: an empty dataserver containing the input variables.
  - **run**: The runner to be used that contains the code and the way to distribute.
  - **nreplica**: an integer specifying the number of trajectories which will be generated.
  - **level**: an integer specifying the number of intervals in each dimension for the grid.
  - **delta**: an integer value to specify the delta parameter, with $\Delta \in [1, level - 1.$

Summary: `TMorris` other methods

- `generateSample`(**Option_t** *option=""): generate the sample used for computing sensitivity screening measures.
  - **option**: no option has been implemented.
- `computeIndexes`(**Option_t** *option=""): compute the sensitivity screening measures.
  - **option**: no option has been implemented.
- void `drawIndexes`(**Option_t** *option=""): draw the Morris measurement. Specific method on top of the generic one from `TSensitivity`.
  - **option**: a string containing the options of the graphics separated by commas.

VI.5 The Sobol method

VI.5.1 Introduction to Sobol's sensitivity indices

The method described in the [30] is, in Uranie, said to be "ï¿½ la Saltelli" in contrast with the other implementation (previously used as default) which is said to be "ï¿½ la Sobol". The difference between the two being the number of assessment used to get a certain precision: for the same results, the implementation "ï¿½ la Sobol" was requesting $n_S(2n_X+2)$ and was offering more numerical results as five algorithms were used. The new implementation "ï¿½ la Saltelli" requests only $n_S(n_X+2)$ estimation but only three algorithms are run. This is summarised as follow where the bold name is the default stored in *--first--* and *--total--*:

a la Saltelli (default method, option="saltelli") First order: Saltelli02 [15], Sumo10 [16], **Martinez11** [19]

Total order: Homma96 [14], Sumo10 [16], **Martinez11** [19]

a la Sobol (option="sobol") First order: **Sobol93** [20], Saltelli02 [15], Jansen99 [25], Sumo10 [16], Martinez11 [19]

Total order: Homma96 [14], Saltelli02 [15], **Jansen99** [25], Sumo10 [16], Martinez11 [19]

Tip
The Martinez11 algorithm is the recommended one, as it provides an estimation of the 95% confidence interval for every coefficient determined.

VI.5.2 Computation of Sobol's sensitivity indices

The **`TSobol`** class computes the total and first order sobol indices using the so-called Saltelli & Tarantola method. It's a Monte-Carlo method which needs only $n_S(n_X+2)$ model evaluations to compute the $2n_X$ indices. In terms of implementation the `TSobol` class differs slightly from the other classes, even though it can be constructed with the three usual cases: a function, an external code or an already filled `TDataServer`. At this stage, `computeIndexes`'s options can be specified to define:

- the way to generate the sample, if needed. Possible choices are: "SRS", "LHS", "QMC=sobol" and "QMC=halton"

- the chosen implementation, to generate the M and N_i matrices (as discussed above and in [30]), possible choices being "sobol" and "saltelli".

If the object is created with an empty `TDataServer` and a code or function, the procedure is the following:

1. `generateSample:` produce the design-of-experiments
2. Run a `TLauncher` object. This has nothing to do with the `TSobol` and is a needed extra step to compute the output corresponding to every design-of-experiments data points.
3. `computeIndexes:` get the indices value.

The following subsections will show an example, detailing every steps one-by-one.

Tip Given the cost of this method (particularly once run with a real code), from version 4.6.0, the way to use and re-use data have be clarified and new methods have been implemented:

- One can re-run an already done estimation. When running `TSobol` with default configuration meaning letting the class generates the design-of-experiments and launching the estimations, there are two files created: `_sobol_sampling_.dat` and `_sobol_launching_.dat`. Both contains input value, internal variable to figure out from what matrices every configuration is taken out of (see [30] for more explanation), but only the latter contains the output values (after estimations). To re-use one of these `_sobol_launching_.dat`, one should simply use the constructor whose only mandatory arguments are the dataserver pointer, the list of input variables (second argument) and the list of output variables (third argument) in the usual form, once the file has been loaded into the dataserver thanks to `fileDataRead`). An example can be found in Section XIV.5.13.
- One can use the "WithData" option which allows to shorten a bit the sobol indices estimation by providing already done estimations. This means that if one has already a design-of-experiments (LHS, SRS or QMC for instance) these computations are done already and will be used as the $2 \times n_S$ first estimation corresponding to both the M and N matrices (see [30] for more details). The class will still have to create all the cross configurations (the N_i matrices) and launch their corresponding estimations. This new (from v4.6.0) option should not be used with a constructor that would not provided a way to perform the estimation. An example can be found in Section XIV.5.14.
- If the statistical accuracy is not sufficient, up to v4.6.0, one has to re-run the macro requesting more computations but losing the ones done previously. A new method has been created in order to be able to use all estimations done in a previous attempt by passing the file `_sobol_launching_.dat`, already introduced above. This method is called `loadOtherSobolFile` and it takes as only argument the name of the input file that will contains just as much attributes as needed for this analysis (inputs and outputs, but also the internal variable to details from which matrices the configurations are coming from). The important matter here is to be sure that the seed used to generate the design-of-experiments in the imported file and the one used to re-generate a second step are not the same. An example can be found in Section XIV.5.15.

VI.5.2.1 Macro computing the Sobol sensitivity indices

The example script below uses the **`TSobol`** class to compute and display Sobol sensitivity indices:

```
{
  gROOT->LoadMacro("UserFunctions.C");

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsflowrate", "DataBase flowrate");
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  Int_t nComp = 3000;
  TSobol * tsobol = new TSobol(tds, "flowrateModel", nComp, "rw:r:tu:tl:hu:hl:l:kw", " ↩
      flowrateModel", "pouet");

  tsobol->computeIndexes();

  TCanvas *cc = new TCanvas("c1", "histgramme",5,64,1270,667);
  tsobol->drawIndexes("Flowrate", "", "nonewcanv,hist,all");

  TCanvas *ccc = new TCanvas("c2", "pie",5,64,1270,667);
  TPad *apad = new TPad("apad","apad",0, 0.03, 1, 1); apad->Draw(); apad->cd();
  tsobol->drawIndexes("Flowrate", "", "nonewcanv,pie");

  }
```

The figures resulting from this script are shown below:

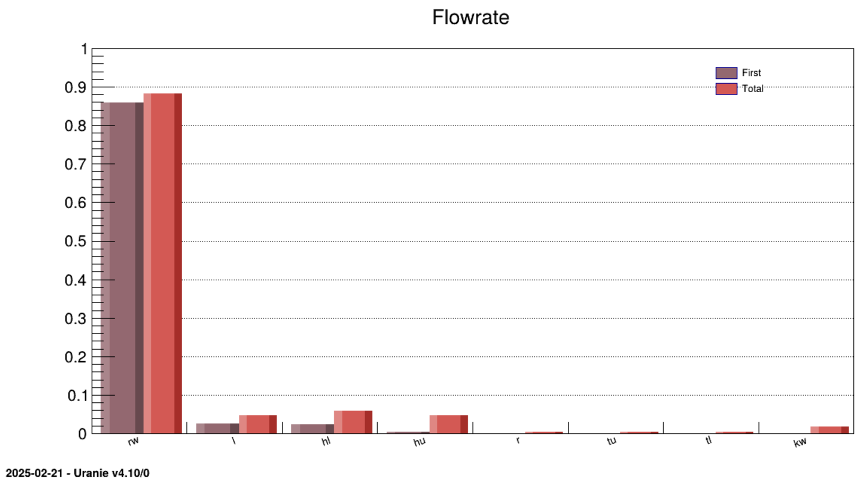


Figure VI.5: Histogram of Sobol's indices

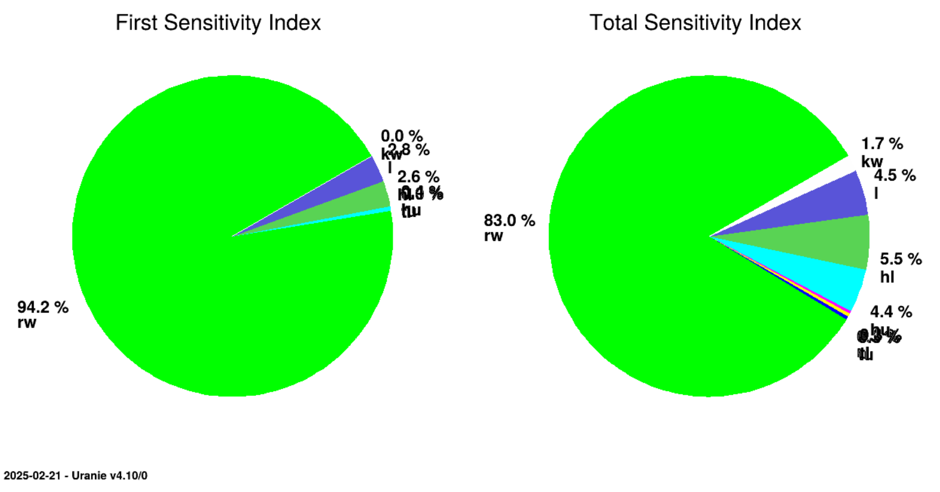


Figure VI.6: Pie chart of Sobol's indices

VI.5.2.2 Specifying the input parameters

First, define the uncertain parameters and add them to a `TDataServer` object:

```
// Define the DataServer
TDataServer *tds = new TDataServer("tdsflowrate", "DataBase flowrate");
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

VI.5.2.3 `TSobol` constructor

There are four different constructors to build a **`TSobol`** object, each corresponding to a different problem:

- the model is an analytic function run by Uranie,
- the model is a code run by Uranie,
- the outputs of the model are already computed and saved in a `TDataServer` object.
- the model is either a function or a code and the problem is specified through a Relauncher architecture.

Warning

The two first request a parameter called n_{Comp}: it represents the number of estimation to be done by the code or the function in total. This value is used to estimate the size of the matrices M and N_i introduced in [30], using the formula

$$n_S = \frac{n_{\text{Comp}}}{n_X + 2}$$

VI.5.2.3.1 `TSobol` constructor for an analytic function

The constructor prototype used with an analytic function is:

```
// Create a TSobol object with an analytic function
TSobol(TDataServer *tds,  void *fcn,  const char *inp,  const char *out, Int_t nComp,   ↩
    Option_t *option="")
TSobol(TDataServer *tds,  const char *fcn,  Int_t nComp,  const char *inp,  const char *out ↩
    , Option_t *option="")
```

This constructor takes five arguments:

- a pointer to a `TDataServer` object,
- a pointer to an analytic function (either a `void` or a `const char` that represents the function's name when it has been loaded in ROOT's memory),
- an integer to specify the number of computation to be performed,
- a string to specify the names of the input factors separated by ':' (ex. "rw:r:tu:tl:hu:hl:l:kw"), its default value is the empty string "",
- a string to specify the names of the output variables of the model, its default value is the empty string "".

Here is an example of how to use the constructor with an analytic function:

```
Int_t ncomp = 100;
TSobol * tsobol = new TSobol(tds, "flowrateModel", ncomp, "rw:r:tu:tl:hu:hl:l:kw", " ↩
    flowrateModel");
```

VI.5.2.3.2 `TSobol` constructor for a code

The constructor prototype used with a code is:

```
// Create a TSobol object with a code
TSobol(TDataServer *tds, TCode *fcode,  Int_t nComp, Option_t *option="")
```

This constructor takes three arguments:

- a pointer to a `TDataServer` object,
- a pointer to a **`TCode`**,
- an integer to specify the number of computations to be performed.

Here is an example of the use of this constructor on the flowrate case:

```
// The reference input file
TString sJDDReference = TString("flowrate_input_with_keys.in");

// Set the reference input file and the key for each input attributes
tds->getAttribute("rw")->setFileKey(sJDDReference, "Rw");
tds->getAttribute("r")->setFileKey(sJDDReference, "R");
tds->getAttribute("tu")->setFileKey(sJDDReference, "Tu");
tds->getAttribute("tl")->setFileKey(sJDDReference, "Tl");
tds->getAttribute("hu")->setFileKey(sJDDReference, "Hu");
tds->getAttribute("hl")->setFileKey(sJDDReference, "Hl");
```

```
tds->getAttribute("l")->setFileKey(sJDDReference, "L");
tds->getAttribute("kw")->setFileKey(sJDDReference, "Kw");

// The output file of the code
TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
// The attribute in the output file
fout->addAttribute(new TAttribute("yhat"));

// Instanciation de mon code
TCode *mycode = new TCode(tds, "flowrate -s -k");
// Adding the outputfile to the tcode object
mycode->addOutputFile( fout );

Int_t nComp = 100;
TSobol * tsobolC = new TSobol(tds, mycode, nComp);
```

VI.5.2.3.3 `TSobol` constructor for a runner

The constructor prototype used with a runner is:

```
// Create a TSobol object with a runner
TSobol(TDataServer *tds, TRun *run,  Int_t nComp, Option_t *option="")
```

This constructor takes three arguments:

- a pointer to a `TDataServer` object,
- a pointer to a **`TRun`**,
- an integer to specify the number of computations to be performed.

Here is an example of the use of this constructor on the flowrate code in a sequential mode:

```
// The input file
TKeyScript infile("flowrate_input_with_keys.in");
// provide the input and their key
infile.setInputs(8, tds->getAttribute("rw"), "Rw", tds->getAttribute("r"), "R",
tds->getAttribute("tu"), "Tu", tds->getAttribute("tl"), "Tl", tds->getAttribute("hu"), "Hu" ↩
    ,
tds->getAttribute("hl"), "Hl", tds->getAttribute("l"), "L", tds->getAttribute("kw"), "Kw");

TAttribute yhat("yhat");
// The output file of the code
TKeyResult outfile("_output_flowrate_withKey_.dat");
// The attribute in the output file
outfile.addOutput(&yhat, "yhat");

// Instanciation de mon code
TCodeEval code("flowrate -s -k");
// Adding the intput/output file to the code
code.addInputFile(&infile);
code.addOutputFile(&outfile);

TSequentialRun run(&code);
run.startSlave();
if(run.onMaster())
{
    Int_t nComp = 100;
    TSobol * tsobolR = new TSobol(tds, &run, nComp);
    // ....
}
```

VI.5.2.3.4 `TSobol` constructor using a filled `TDataServer` object

The constructor prototype used with a `TDataServer` object already containing the simulations is:

```
// Create a TSobol object with already filled TDS
TSobol(TDataServer *tds, const char *inp, const char *out, Option_t *option="")
```

This constructor takes three arguments:

- a pointer to a `TDataServer` object filled,
- a string to specify the names of the input factors separated by ':' (ex. "rw:r:tu:tl:hu:hl:l:kw"),
- a string to specify the names of the output variables of the model.

Below is an example of the constructor with a `TDataServer` object filled:

```
TDataServer *tds = new TDataServer();
tds->fileDataRead("_sobol_launching.dat");
TSobol * tsobol = new TSobol(tds, "rw:r:tu:tl:hu:hl:l:kw", "flowrateModel");
```

Warning
This constructor uses a `TDataServer` object already filled with a specific internal variable (sobol__n__iter__tdsflowreate) and a specific sample!
There are several conditions to use it:

- use the constructor without argument for the `TDataServer`;
- the input factors sample must have been generated with the method `TSobol::generateSample`.

VI.5.2.4 Generating the sample

To generate the Sobol sample, use the `generateSample` method:

```
tsobol->generateSample();
```

Then, the sample generated can be exported in a file and used outside of Uranie to compute the simulations associated.

VI.5.2.5 Computing the indices

To compute the indices, run the method `computeIndexes`:

```
tsobol->computeIndexes();
```

Note that this method is all inclusive: it constructs the sample, launches the simulations and computes the indices.

VI.5.2.6 Displaying the indices

To display graphically the coefficients, use the method `drawIndexes`:

```
void TSensitivity::drawIndexes(TString sTitre, const char *select, Option_t * option)
```

The method needs:

- a `TString` containing the title of the figure,
- a string containing a selection (empty if no selection),
- a string containing the options of the graphics separated by commas.

Some of the options available are:

- "nonewcanv": to not create a new canvas,
- "pie": to display a pie chart,
- "hist": to display a histogram,
- "first": to display the indices of the first order (with "hist" only),
- "total": to display the total indices (with "hist" only),
- "all": to display the total and first order indices (with "hist" only),

In our example the use of this method is:

```
TCanvas *cc = new TCanvas("c1", "histgramme",5,64,1270,667);
cc->Divide(2,1);
cc->cd(1);
tsobol->drawIndexes("Flowrate", "", "nonewcanv,hist,all");
cc->Print("appliUranieFlowrateSobol100Histogram.png");

cc->cd(2);
tsobol->drawIndexes("Flowrate", "", "nonewcanv,pie");
cc->Print("appliUranieFlowrateSobol100Pie.png");
```

VI.5.2.7 Extracting the coefficients

The coefficients, once computed, are stored in a `TTree`. To get this `TTree`, use the method `TSensitivity::getResul`

```
TTree * results = tsobol->getResultTuple();
```

Several methods exist in ROOT to extract data from a `TTree`, it is advised to look for them into the ROOT documentation. We propose two ways of extracting the value of each coefficient from the `TTree`.

VI.5.2.7.1 First method of extraction

The first method use the method `getValue` of the `TSobol` object specifying the order of the extract value ("First" ou "Total"), the related input and possibly more selected options (for example "Algo==\"homa96\"").

```
double hl_first_index = tsobol->getValue("First","hl");
double hl_total_index = tsobol->getValue("Total","hl");
```

VI.5.2.7.2 Second method of extraction

The seond method uses 3 steps to extract an index:

- scan the `TTree` for the chosen input variable (with a selection) in order to obtain its row number. In our example, if we chose the variable "hl", we'll use the command:

  ```
  results->Scan("*","Inp==\"hl\"");
  ```

  and in the resulting figure below, we see that the first order index of "hl" is in the row 40:

  ```
  ***********************************************************************************
  *    Row   *   Out.Out * Inp. * Order. * Method * Algo.Algo * Value * CILow * CIUpp *
  ***********************************************************************************
  *       40 * flowrateM *   hl *  First *  Sobol * --first-- * 0.023 *     0 * 0.136 *
  *       41 * flowrateM *   hl *  First *  Sobol * 02saltell * 0.045 *    -1 *    -1 *
  *       42 * flowrateM *   hl *  First *  Sobol *    sumo10 * 0.020 *    -1 *    -1 *
  *       43 * flowrateM *   hl *  First *  Sobol * martinez1 * 0.023 *     0 * 0.136 *
  *       44 * flowrateM *   hl *  Total *  Sobol * --total-- * 0.058 * 0.046 * 0.072 *
  *       45 * flowrateM *   hl *  Total *  Sobol *   homma96 * 0.078 *    -1 *    -1 *
  *       46 * flowrateM *   hl *  Total *  Sobol *    sumo10 * 0.071 *    -1 *    -1 *
  *       47 * flowrateM *   hl *  Total *  Sobol * martinez1 * 0.058 * 0.046 * 0.072 *
  ***********************************************************************************
  ```

- set the entry of the `TTree` on this row with the method `GetEntry`;

- get the value of the index with `GetValue` method on the "Value" leaf of the `TTree`.

Below is an example of extraction of the index for "hl" in our flowrate case:

```
results->Scan("*","Inp==\"hl\"");
results->GetEntry(40);
Double_t S_Rw_Indexe =  results->GetLeaf("Value")->GetValue();
```

VI.5.2.7.3 Third method of extraction

The third method uses 2 steps to extract an index:

- use the `Draw` method with a selection to select the index, for example the selection for the first order index of "rw" is "Inp==\"rw\" && Algo==\"--first--\"";

- get the pointer on the value of the index with `GetV1` method on the `TTree`.

Below is another example of extraction of the first order index for "rw" in our flowrate case:

```
results->Draw("Value","Inp==\"rw\" && Algo==\"--first--\"","goff");
Double_t S_Rw_IndexeS = results->GetV1()[0];
```

Summary: `TSobol` constructors

- `TSobol`(**`TDataServer`** *tds, **const char** *fcn, **Int_t** nComp, **const char** *inp, **const char** *out, **Option_t** *option="")
  - **tds**: an empty dataserver containing the input variables.
  - **fcn**: the function to analyse (a name).
  - **nComp**: an integer to specify the number of simulations.
  - **inp**: input attribute names separated by colons. They represent all the inputs of the problem. They must refer to existing attributes in the "tds".
  - **out**: attribute names which represent the response of the problem. it must refer to existing attributes in the "tds".
  - **option**: string specifying the sampler method.
- `TSobol`(**`TDataServer`** *tds, **void** *fcn(**double ***,**double ***), **const char** *inp, **const char** *out, **Int_t** nComp, **Option_t** *option="")
  - **tds**: an empty dataserver containing the input variables.
  - **fcn**: the function to analyse (a pointer).
  - **inp**: input attribute names separated by colons. They represent all the inputs of the problem. They must refer to existing attributes in the "tds".
  - **out**: attribute names which represent the response of the problem. it must refer to existing attributes in the "tds".
  - **nComp**: an integer to specify the number of simulations.
  - **option**: string specifying the sampler method.
- `TSobol`(**`TDataServer`** *tds, **const char** *inp, **const char** *out, **Option_t** *option="")
  - **tds**: a dataserver containing the input and output attributes and their associated data.
  - **inp**: input attribute names separated by colons. They represent all the inputs of the problem. They must refer to existing attributes in the "tds".
  - **out**: attribute names which represent the response of the problem. it must refer to existing attributes in the "tds".
  - **option**: string specifying the sampler method.
- `TSobol`(**`TDataServer`** *tds, **`TCode`** *fcode, **Int_t** ns, **Option_t** *option="")
  - **tds**: an empty dataserver containing the input variables.
  - **fcode**: the code to analyse.
  - **nComp**: an integer to specify the number of simulations.
  - **option**: string specifying the sampler method.

Summary: `TSobol` other methods

- `generateSample`(**Option_t** *option=""): generate the sample used for computing the sensitivity indices.
  - **option**: no option has been implemented (Default = "").
- `computeIndexes`(**Option_t** *option=""): compute the Sobol sensitivity indices.
  - **option**: no option has been implemented.
- void `drawIndexes`(**`TString`** sTitre, **const char*** select="", **Option_t** *option=""): draw the coefficients.
  - **sTitre**: a string containing the title of the figure.
  - **select**: a string containing a solution.
  - **option**: a string containing the options of the graphics separated by commas.

VI.6 Fourier-based methods

Warning
These methods require the FFTW prerequisite (as discussed in Section I.1.2.2).

VI.6.1 Introducing the method

This section is a short excerpt of [30]

VI.6.1.1 The FAST method

The *Fourier Amplitude Sensitivity Test* (FAST) [26, 27] is a procedure that provides a way to estimate the expected value and variance of the output variable of a model, along with the contribution of the input factors to this variance. An advantage of it, is that the evaluation of sensitivity can be carried out independently for each factor using just a set of runs because all the terms in a Fourier expansion are mutually orthogonal. The main idea behind this procedure is to transform the n_X-dimensional integration into a single-dimension one, by using the transformation

$$X_i = G_i(\sin(\omega_i \times s)),$$

where ideally, $\{\omega_i\}$ is a set of angular frequencies said to be incommensurate (meaning that no frequency can be obtained by linear combination of the other ones when using integer coefficients) and G_i is a transformation function chosen in order to ensure that the variable is sampled accordingly with the probability density function of X_i (meaning that they are all uniformly distributed in their respective volume definition).

The first order coefficient is then obtained by estimating the variance for a fundamental ω_i and its harmonics. The important point to notice, for a real computation, is the limitation of the sum that, in the previous equation, runs up to infinity. A truncation is done by imposing a cut-off with a factor M called the **interference factor** (whose default value in Uranie is set to 6).

VI.6.1.2 The RBD method

The *Random Balance Design* (RBD) [28] method selects n_S design points over a curve in the input space. The input space is explored here using the same frequency ω. However the curve is not space-filling, therefore, we take random permutations of the coordinates of such points, to generate a set of scrambled points that cover the input space. The model is then evaluated at each design point. Subsequently, the model outputs are re-ordered such that the design points are in increasing order with respect to factor X_i.

Thanks to the use of permutations, the total cost is of the order of n_S assessments instead of the order of $n_S \times n_X$ for the FAST one.

VI.6.2 Implementation of methods

Both `TFAST` and `TRBD` can be constructed either from an external code or from a function. In the implementation done within Uranie there are several modifiable parameters that can be considered before starting an analysis using the FAST method:

- The transformation function G_i chosen among the following list:
  - Cukier: $X_i = \bar{X}_i \exp(\bar{\nu}_i \sin(\omega_i s))$
  - SaltelliA: $X_i = 0.5 + \frac{1}{\pi}\arcsin(\sin(\omega_i s))$
  - SaltelliB: $X_i = 0.5 + \frac{1}{\pi}\arcsin(\sin(\omega_i s + \phi_i))$

  In this list, $\bar{X}_i$ is the nominal value of the factor X_i, $\bar{\nu}_i$ denotes the endpoints that define the estimated range of uncertainty of X_i, ϕ_i is a random phase shift taken value in $[0, 2\pi]$ and s evolves in $[-\pi/2, \pi/2]$.
- The interference factor: M can be changed as well.
- The frequencies: by providing a vector, it is possible to set a default at the frequencies' value used instead of having them determined by a specific algorithm to avoid, as best as possible, the interference.

The only common parameter changeable for both methods (and directly in the construction) is the number of samples. Once the object is constructed, running the `computeIndexes` will compute the sensitivity indices and the usual drawing methods will allow to retrieve and represent the results, as already explained for other methods.

Warning
If the Fourier-based object is constructed to run a function, the output used for the estimation will be the first output provided by the function, unless the sixth argument of the constructor is filled.

VI.6.3 Computation of Sobol indices with the FAST method

In Uranie, computing sensitivity with the FAST method is dealt with the eponymous class `TFast` which inherits from the **`TSensitivity`** class.

This class handles all the steps to compute the Sobol indices:

- generating the deterministic sample;
- running the code or the analytic function to get the response on the sample;
- computing the first order index for each input variable.

VI.6.3.1 Example: simple computation of sensitivity indices with FAST

The example script below uses the `TFast` class to compute and display Sobol sensitivity indices:

```
{
  gROOT->LoadMacro("UserFunctions.C");

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  // \param Size of a sampling.
  Int_t nS = 500;
  // Graph
  TFast * tfast = new TFast(tds, "flowrateModel", nS);
  tfast->computeIndexes("graph");

  TCanvas *cc = new TCanvas("canhist", "histgramme",1);
  tfast->drawIndexes("Flowrate", "", "nonewcanv,hist,first");

  TCanvas *ccc = new TCanvas("canpie", "TPie",1);
  TPad *apad = new TPad("apad","apad",0, 0.03, 1, 1); apad->Draw(); apad->cd();
  tfast->drawIndexes("Flowrate", "", "nonewcanv,pie,first");

  }
```

The figures resulting from this script are shown below, the first shows the frequencies:

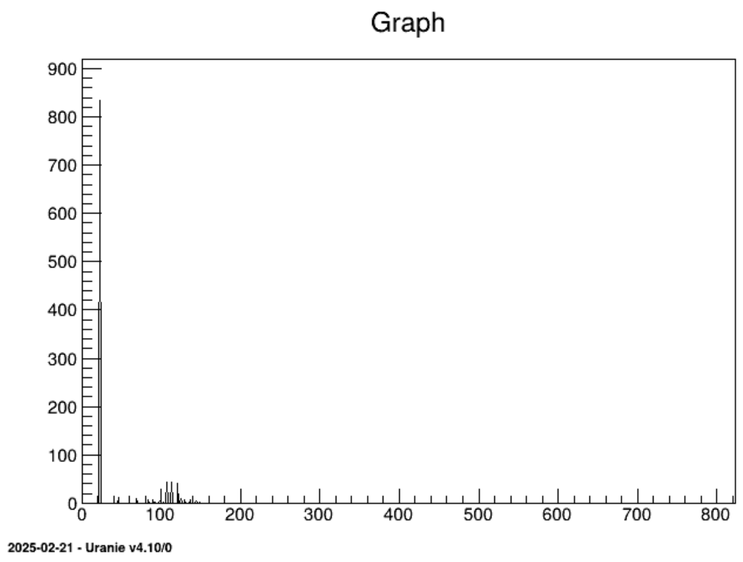


Figure VI.7: Frequency spectrum from the FAST estimation

The two figures below display the sensitivity indices in a histogram and in a pie chart:

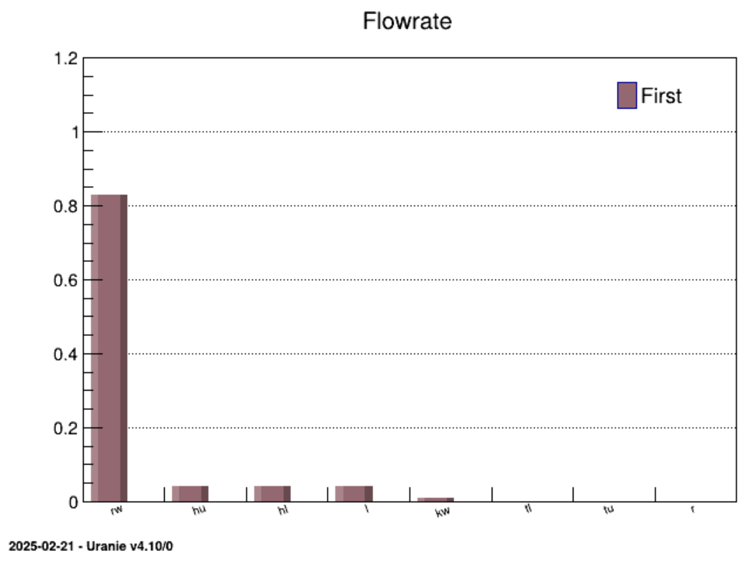


Figure VI.8: Histogram of FAST's indices

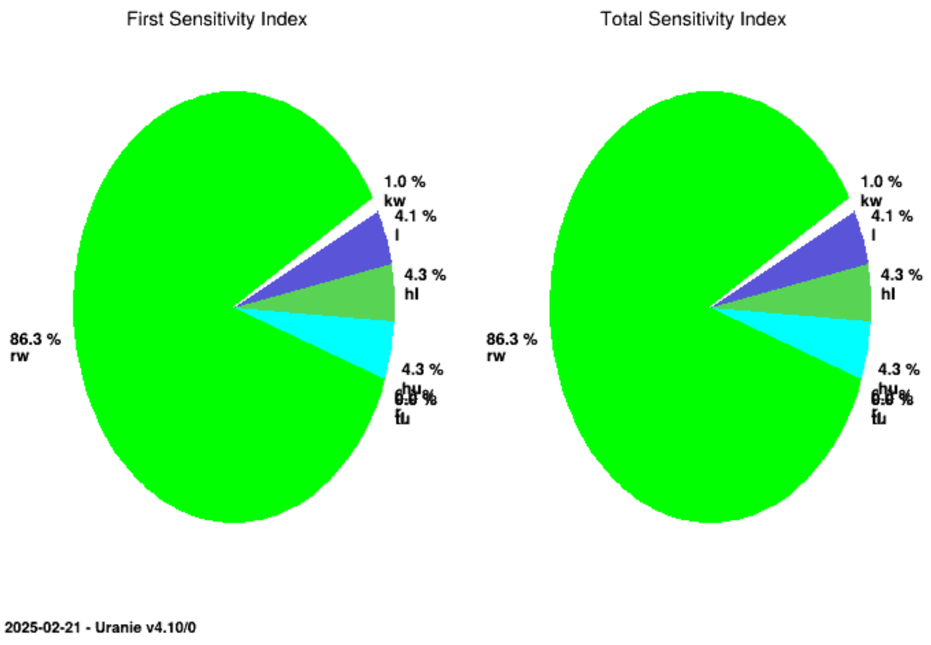


Figure VI.9: Pie chart of FAST's indices

VI.6.3.2 Specifying the input parameters

First, define the uncertain parameters and add them to a `TDataServer` object:

```
// Define the DataServer
TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
```

```
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

VI.6.3.3 **TFast** constructor

There are three different constructors to build a **TFast** object, each corresponding to a different problem:

- the model is an analytic function run by Uranie,
- the model is a code run by Uranie.
- the model is either a function or a code and the problem is specified through a Relauncher architecture.

VI.6.3.3.1 **TFast** constructor for an analytic function

The constructor prototype used with an analytic function is:

```
// Create a TFast object with an analytic function
TFast(TDataServer *tds,  void *fcn(double*,double*),  const char *inp,  const char *out,  ↩
    Int_t ns)
TFast(TDataServer *tds,  const char *fcn,  Int_t ns,  const char *inp="",  const char *out= ↩
    "", Option_t *option="")
```

This constructor takes five arguments:

- a pointer to a `TDataServer` object,
- a pointer to an analytic function (either a `void` or a `const char` that represents the function's name when it has been loaded in ROOT's memory),
- an integer to specify the number of simulations, its default value is 100,
- a string to specify the names of the input factors separated by ':' (ex. "rw:r:tu:tl:hu:hl:l:kw"), its default value might be the empty string "",
- a string to specify the names of the output variables of the model, its default value might be the empty string "".

Here is an example of how to use the constructor with an analytic function:

```
Int_t ns = 50;
TFast * tfast = new TFast(tds, "flowrateModel", ns, "rw:r:tu:tl:hu:hl:l:kw", "flowrateModel ↩
    ");
```

VI.6.3.3.2 **TFast** constructor for a code

The constructor prototype used with a code is:

```
// Create a TFast object with a code
TFast(TDataServer *tds, TCode *fcode,  Int_t ns, Option_t *option="")
```

This constructor takes three arguments:

- a pointer to a `TDataServer` object,
- a pointer to a **`TCode`**,
- an integer to specify the number of simulations.

Here is an example of the use of this constructor on the flowrate case:

```
// The reference input file
TString sJDDReference = TString("flowrate_input_with_keys.in");

// Set the reference input file and the key for each input attributes
tds->getAttribute("rw")->setFileKey(sJDDReference, "Rw");
tds->getAttribute("r")->setFileKey(sJDDReference, "R");
tds->getAttribute("tu")->setFileKey(sJDDReference, "Tu");
tds->getAttribute("tl")->setFileKey(sJDDReference, "Tl");
tds->getAttribute("hu")->setFileKey(sJDDReference, "Hu");
tds->getAttribute("hl")->setFileKey(sJDDReference, "Hl");
tds->getAttribute("l")->setFileKey(sJDDReference, "L");
tds->getAttribute("kw")->setFileKey(sJDDReference, "Kw");

// The output file of the code
TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
// The attribute in the output file
fout->addAttribute(new TAttribute("yhat"));

// Instanciation de mon code
TCode *mycode = new TCode(tds, "flowrate -s -k");
// Adding the outputfile to the tcode object
mycode->addOutputFile( fout );

// Size of a sampling.
Int_t nS = 50;
TFast * tfastC = new TFast(tds, mycode, nS);
```

VI.6.3.3.3 `TFast` constructor for a runner

The constructor prototype used with a runner is:

```
// Create a TFast object with a runner
TFast(TDataServer *tds, TRun *run,  Int_t ns, Option_t *option="")
```

This constructor takes three arguments:

- a pointer to a `TDataServer` object,
- a pointer to a **`TRun`**,
- an integer to specify the number of simulations.

Here is an example of the use of this constructor on the flowrate code in a sequential mode:

```
// The input file
TKeyScript infile("flowrate_input_with_keys.in");
// provide the input and their key
infile.setInputs(8, tds->getAttribute("rw"), "Rw", tds->getAttribute("r"), "R",
tds->getAttribute("tu"), "Tu", tds->getAttribute("tl"), "Tl", tds->getAttribute("hu"), "Hu" ↩
    ,
tds->getAttribute("hl"), "Hl", tds->getAttribute("l"), "L", tds->getAttribute("kw"), "Kw");
```

```
TAttribute yhat("yhat");
// The output file of the code
TKeyResult outfile("_output_flowrate_withKey_.dat");
// The attribute in the output file
outfile.addOutput(&yhat, "yhat");

// Instanciation de mon code
TCodeEval code("flowrate -s -k");
// Adding the intput/output file to the code
code.addInputFile(&infile);
code.addOutputFile(&outfile);

TSequentialRun run(&code);
run.startSlave();
if(run.onMaster())
{
    Int_t nS = 50;
    TFast * tfastR = new TFast(tds, &run, nS);
    //...
}
```

VI.6.3.4 Generating the sample

To generate the FAST sample, use the `generateSample` method:

```
tfast->generateSample();
```

Then, the sample generated can be exported in a file and used outside of Uranie to compute the simulations associated.

VI.6.3.5 Computing the indices

To compute the indices, run the method `computeIndexes`:

```
 tfast->computeIndexes();
```

Note that this method is all inclusive: it constructs the sample, launches the simulations and computes the indices. If the "graph" option is provided, a graph of frequency will be drawn, as the one shown in Figure VI.7

VI.6.3.6 Displaying the indices

To display graphically the coefficients, use the method `drawIndexes`:

```
void TSensitivity::drawIndexes(TString sTitre, const char *select, Option_t * option)
```

The method needs:

- a `TString` containing the title of the figure,
- a string containing a selection (empty if no selection),
- a string containing the options of the graphics separated by commas.

Some of the options available are:

- "nonewcanv": to not create a new canvas,
- "pie": to display a pie chart,
- "hist": to display a histogram,
- "first": to display the indices of the first order (with "hist" only),

In our example the use of this method is:

```
TCanvas *cc = new TCanvas("canhist", "histgramme");
tfast->drawIndexes("Flowrate", "", "nonewcanv,hist,first");
cc->Print("appliUranieFlowrateFAST100Histogram.png");

TCanvas *ccc = new TCanvas("canpie", "TPie");
tfast->drawIndexes("Flowrate", "", "nonewcanv,pie");
ccc->Print("appliUranieFlowrateFAST100Pie.png");
```

VI.6.3.7 Extracting the coefficients

The coefficients, once computed, are stored in a `TTree`. To get this `TTree`, use the method `TSensitivity::getResul`

```
TTree * results = tfast->getResultTuple();
```

Several methods exist in ROOT to extract data from a `TTree`, it is advised to look for them into the ROOT documentation. We propose two ways of extracting the value of each coefficient from the `TTree`.

VI.6.3.7.1 First method of extraction

The first method uses 3 steps to extract an index:

- scan the `TTree` for the chosen input variable (with a selection) in order to obtain its row number. In our example, if we chose the variable "hl", we'll use the command:

  ```
  results->Scan("*","Inp==\"hl\"");
  ```

  and in the resulting figure below we see that the first order index of "hl" is in the row 10:

  ```
  ************************************************************************
  *    Row   *       Out *  Inp *  Order * Method *      Algo *     Value *
  ************************************************************************
  *       10 * flowrateM *   hl *  First *   FAST * --first-- * 0.0413399 *
  *       11 * flowrateM *   hl *  Total *   FAST * --total-- * 0.0413399 *
  ************************************************************************
  ```

- set the entry of the `TTree` on this row with the method `GetEntry`;
- get the value of the index with `GetValue` method on the "Value" leaf of the `TTree`.

Below is an example of extraction of the index for "hl" in our flowrate case:

```
results->Scan("*","Inp==\"hl\"");
results->GetEntry(10);
Double_t S_Rw_Indexe =  results->GetLeaf("Value")->GetValue();
```

VI.6.3.7.2 Second method of extraction

The second method uses 2 steps to extract an index:

- use the `Draw` method with a selection to select the index, for example the selection for the first order index of "rw" is "Inp==\"rw\" && Algo==\"--first--\"";
- get the pointer on the value of the index with `GetV1` method on the `TTree`.

Below is another example of extraction of the first order index for "rw" in our flowrate case:

```
results->Draw("Value","Inp==\"rw\" && Algo==\"--first--\"","goff");
Double_t S_Rw_IndexeS = results->GetV1()[0];
```

Summary: `TFast` object

- `TFast`(**`TDataServer`** *tds, **const char** *fcn, **Int_t** ns=100, **const char** *inp="", **const char** *out="")
  - **tds**: an empty dataserver containing the input variables.
  - **fcn**: the function to analyse (the function's name).
  - **ns**: an integer to specify the number of simulations.
  - **out**: attribute names which represent the responses of the problem. it must refer to existing attributes in the "tds".
  - **inp**: input attribute names separated by colons. They represent all the inputs of the problem. They must refer to existing attributes in the "tds".
- `TFast`(**`TDataServer`** *tds, **void** *fcn(**double** *,**double** *), **const char** *inp, **const char** *out, **Int_t** ns=100)
  - **tds**: an empty dataserver containing the input variables.
  - **fcn**: the function to analyse (a pointer).
  - **ns**: an integer to specify the number of simulations.
  - **out**: attribute names which represent the responses of the problem. it must refer to existing attributes in the "tds".
  - **inp**: input attribute names separated by colons. They represent all the inputs of the problem. They must refer to existing attributes in the "tds".
- `TFast`(**`TDataServer`** *tds, **`TCode`** *fcode, **Int_t** ns)
  - **tds**: an empty dataserver containing the input variables.
  - **fcode**: the code to analyse .
  - **ns**: an integer to specify the number of simulations.
- `TFast`(**`TDataServer`** *tds, **`TRun`** *run, **Int_t** ns)
  - **tds**: an empty dataserver containing the input variables.
  - **run**: the runner to use the code/function to analyse.
  - **ns**: an integer to specify the number of simulations.
- `generateSample`(**Option_t** *option=""): generate the sample used for computing the sensitivity indices.
  - **option**: no option has been implemented.
- `computeIndexes`(**Option_t** *option=""): compute the Sobol sensitivity indices.
  - **option**: string specifying the drawIndexes option used.
- void `drawIndexes`(**`TString`** sTitre, **const char*** select="", **Option_t** *option=""): draw the coefficients.
  - **sTitre**: a string containing the title of the figure.
  - **select**: a string containing a solution.
  - **option**: a string containing the options of the graphics separated by commas.

VI.6.4 Computation of Sobol indices with the method RBD

In Uranie, computing sensitivity with the method RBD is dealt with the eponymous class `TRBD` which inherits from the **`TSensitivity`** class and the **`TFast`**.

This class handles all the steps to compute the Sobol indices:

- generating the deterministic sample;
- running the code or the analytic function to get the response on the sample;
- computing the first order index for each input variable.

VI.6.4.1 Example: simple computation of sensitivity indices with RBD

The example script below uses the `TRBD` class to compute and display Sobol sensitivity indices:

```
{
 gROOT->LoadMacro("UserFunctions.C");

 // Define the DataServer
 TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
 tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
 tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
 tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
 tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
 tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
 tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
 tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
 tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

 // Size of a sampling.
 Int_t nS = 500;
 // Graph
 TRBD * trbd = new TRBD(tds, "flowrateModel", nS);
 trbd->computeIndexes("graph");

 TCanvas *cc = new TCanvas("canhist", "histogramme",1);
 trbd->drawIndexes("Flowrate", "", "nonewcanv,hist,first");

 TCanvas *ccc = new TCanvas("canpie", "TPie",1);
 TPad *apad = new TPad("apad","apad",0, 0.03, 1, 1); apad->Draw(); apad->cd();
 trbd->drawIndexes("Flowrate", "", "nonewcanv,pie");

  }
```

The figures resulting from this script are shown below, the first shows the frequencies:

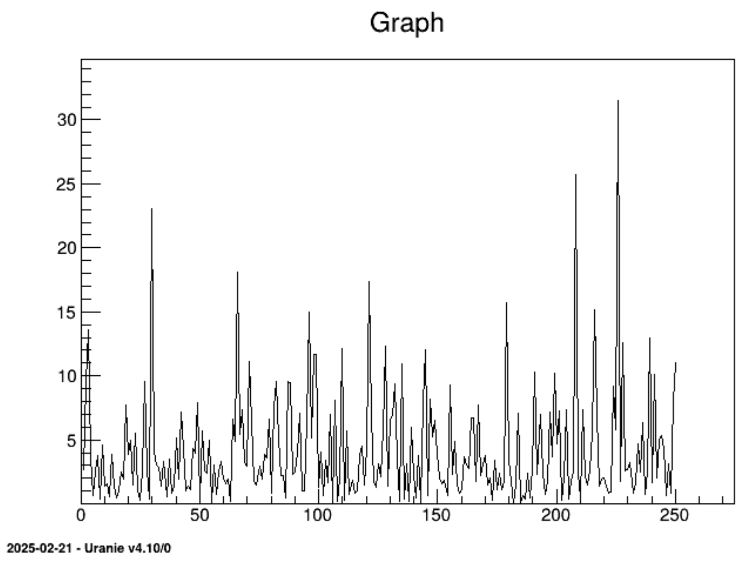


Figure VI.10: Frequency spectrum from the RBD estimation

The two figures below display the sensitivity indices in a histogram and in a pie chart:

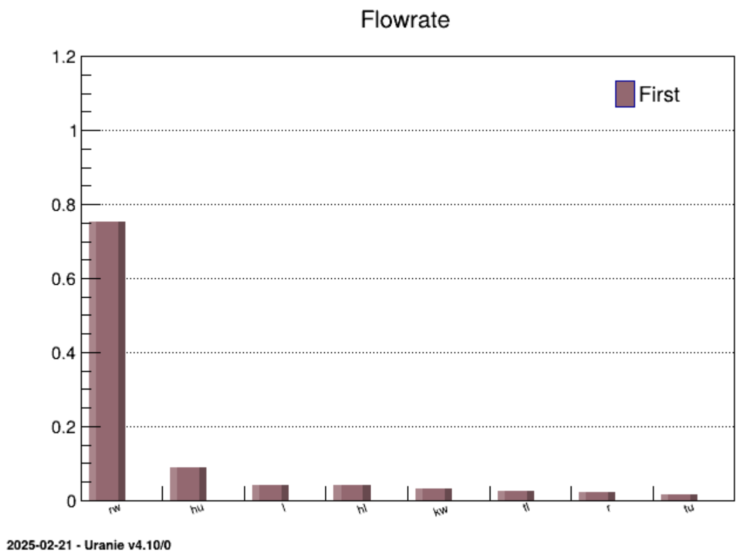


Figure VI.11: Histogram of RBD's indices

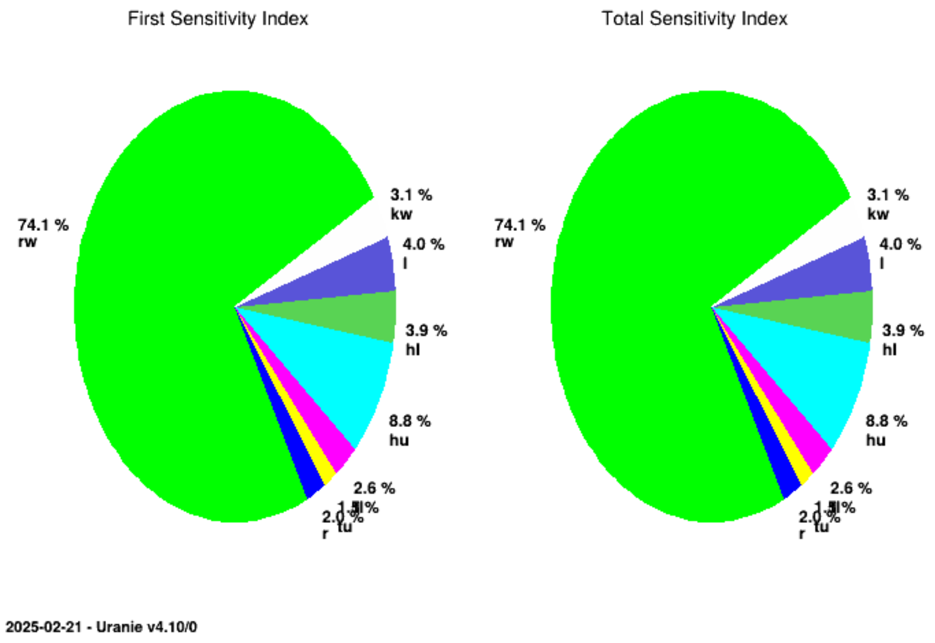


Figure VI.12: Pie chart of RBD's indices

VI.6.4.2 Specifying the input parameters

First, define the uncertain parameters and add them to a `TDataServer` object:

```
// Define the DataServer
TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

VI.6.4.3 **`TRBD`** constructor

There are three different constructors to build a **`TRBD`** object, each corresponding to a different problem:

- the model is an analytic function run by Uranie,
- the model is a code run by Uranie.
- the model is either a function or a code and the problem is specified through a Relauncher architecture.

VI.6.4.3.1 **`TRBD`** constructor for an analytic function

The constructor prototype used with an analytic function is:

```
// Create a TRBD object with an analytic function
TRBD(TDataServer *tds, void *fcn, const char *inp, const char *out, Int_t ns)
TRBD(TDataServer *tds, const char *fcn, Int_t ns, const char *inp="", const char *out="↩
    ")
```

This constructor takes five arguments:

- a pointer to a `TDataServer` object,
- a pointer to an analytic function (either a `void` or a `const char` that represents the function's name when it has been loaded in ROOT's memory),
- an integer to specify the number of simulations, its default value is 100,
- a string to specify the names of the input factors separated by ':' (ex. "rw:r:tu:tl:hu:hl:l:kw"), its default value might be the empty string "",
- a string to specify the names of the output variables of the model, its default value might be the empty string "".

Here is an example of how to use the constructor with an analytic function:

```
Int_t nS = 50;
TRBD * trbd = new TRBD(tds, "flowrateModel", nS, "rw:r:tu:tl:hu:hl:l:kw", "flowrateModel");
```

VI.6.4.3.2 `TRBD` constructor for a code

The constructor prototype used with a code is:

```
// Create a TRBD object with a code
TRBD(TDataServer *tds, TCode *fcode, Int_t ns, Option_t *option="")
```

This constructor takes three arguments:

- a pointer to a `TDataServer` object,
- a pointer to a **`TCode`**,
- an integer to specify the number of simulations.

Here is an example of the use of this constructor on the flowrate case:

```
// The reference input file
TString sJDDReference = TString("flowrate_input_with_keys.in");

// Set the reference input file and the key for each input attributes
tds->getAttribute("rw")->setFileKey(sJDDReference, "Rw");
tds->getAttribute("r")->setFileKey(sJDDReference, "R");
tds->getAttribute("tu")->setFileKey(sJDDReference, "Tu");
tds->getAttribute("tl")->setFileKey(sJDDReference, "Tl");
tds->getAttribute("hu")->setFileKey(sJDDReference, "Hu");
tds->getAttribute("hl")->setFileKey(sJDDReference, "Hl");
tds->getAttribute("l")->setFileKey(sJDDReference, "L");
tds->getAttribute("kw")->setFileKey(sJDDReference, "Kw");

// The output file of the code
TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
```

```
// The attribute in the output file
fout->addAttribute(new TAttribute("yhat"));

// Instanciation de mon code
TCode *mycode = new TCode(tds, "flowrate -s -k");
// Adding the outputfile to the tcode object
mycode->addOutputFile( fout );

// Size of a sampling.
Int_t nS2 = 500;
TRBD * trbd2 = new TRBD(tds, mycode, nS2);
```

VI.6.4.3.3 **TRBD** constructor for a runner

The constructor prototype used with a runner is:

```
// Create a TRBD object with a runner
TRBD(TDataServer *tds, TRun *frun,  Int_t ns, Option_t *option="")
```

This constructor takes three arguments:

- a pointer to a `TDataServer` object,
- a pointer to a **TRun**,
- an integer to specify the number of simulations.

Here is an example of the use of this constructor on the flowrate code in sequential mode:

```
// The input file
TKeyScript infile("flowrate_input_with_keys.in");
// provide the input and their key
infile.setInputs(8, tds->getAttribute("rw"), "Rw", tds->getAttribute("r"), "R",
tds->getAttribute("tu"), "Tu", tds->getAttribute("tl"), "Tl", tds->getAttribute("hu"), "Hu" ↩
    ,
tds->getAttribute("hl"), "Hl", tds->getAttribute("l"), "L", tds->getAttribute("kw"), "Kw");

TAttribute yhat("yhat");
// The output file of the code
TKeyResult outfile("_output_flowrate_withKey_.dat");
// The attribute in the output file
outfile.addOutput(&yhat, "yhat");

// Instanciation de mon code
TCodeEval code("flowrate -s -k");
// Adding the intput/output file to the code
code.addInputFile(&infile);
code.addOutputFile(&outfile);

TSequentialRun run(&code);
run.startSlave();
if(run.onMaster())
{
    // Size of a sampling.
    Int_t nS2 = 500;
    TRBD * trbdR = new TRBD(tds, &run, nS2);
    //...
}
```

VI.6.4.4 Generating the sample

To generate the RBD sample, use the `generateSample` method:

```
trbd->generateSample();
```

Then, the sample generated can be exported in a file and used outside of Uranie to compute the simulations associated.

VI.6.4.5 Computing the indices

To compute the indices, run the method `computeIndexes`:

```
trbd->computeIndexes();
```

Note that this method is all inclusive: it constructs the sample, launches the simulations and computes the indices. If the "graph" option is provided, a graph of frequency will be drawn, as the one shown in Figure VI.10

VI.6.4.6 Displaying the indices

To display graphically the coefficients, use the method `drawIndexes`:

```
void TSensitivity::drawIndexes(TString sTitre, const char *select, Option_t * option)
```

The method needs:

- a `TString` containing the title of the figure,
- a string containing a selection (empty if no selection),
- a string containing the options of the graphics separated by commas.

Some of the options available are:

- "nonewcanv": to not create a new canvas,
- "pie": to display a pie chart,
- "hist": to display a histogram,
- "first": to display the indices of the first order (with "hist" only),
- ...

In our example the use of this method is:

```
TCanvas *cc = new TCanvas("canhist", "histogramme");
trbd->drawIndexes("Flowrate", "", "nonewcanv,hist,first");
cc->Print("appliUranieFlowrate_RBD_50Histogram.png");

TCanvas *ccc = new TCanvas("canpie", "TPie",5,64,1270,560);
trbd->drawIndexes("Flowrate", "", "nonewcanv,pie");
ccc->Print("appliUranieFlowrate_RBD_50Pie.png");
```

VI.6.4.7 Extracting the coefficients

The coefficients, once computed, are stored in a `TTree`. To get this `TTree`, use the method `TSensitivity::getResultTup`

```
TTree * results = trbd->getResultTuple();
```

Several methods exist in ROOT to extract data from a `TTree`, it is advised to look for them into the ROOT documentation. We propose two ways of extracting the value of each coefficient from the `TTree`.

VI.6.4.7.1 First method of extraction

The first method uses 3 steps to extract an index:

- scan the `TTree` for the chosen input variable (with a selection) in order to obtain its row number. In our example, if we chose the variable "hl", we'll use the command:

  ```
  results->Scan("*","Inp==\"hl\"");
  ```

  and in the resulting figure below we see that the first order index of "hl" is in the row 10:

  ```
  ***********************************************************************
  *    Row   *       Out *  Inp *  Order * Method *      Algo *     Value *
  ***********************************************************************
  *       10 * flowrateM *   hl *  First *    RBD * --first-- * 0.0392761 *
  *       11 * flowrateM *   hl *  Total *    RBD * --total-- * 0.0392761 *
  ***********************************************************************
  ```

- set the entry of the `TTree` on this row with the method `GetEntry`;
- get the value of the index with `GetValue` method on the "Value" leaf of the `TTree`.

Below is an example of extraction of the index for "hl" in our flowrate case:

```
results->Scan("*","Inp==\"hl\"");
results->GetEntry(100);
Double_t S_Rw_Indexe =  results->GetLeaf("Value")->GetValue();
```

VI.6.4.7.2 Second method of extraction

The second method uses 2 steps to extract an index:

- use the `Draw` method with a selection to select the index, for example the selection for the first order index of "rw" is "Inp==\"rw\" && Algo==\"--first--\"";
- get the pointer on the value of the index with `GetV1` method on the `TTree`.

Below is another example of extraction of the first order index for "rw" in our flowrate case:

```
results->Draw("Value","Inp==\"rw\" && Algo==\"--first--\"","goff");
Double_t S_Rw_IndexeS = results->GetV1()[0];
```

Summary: `TRBD` object

- `TRBD`(**`TDataServer`** *tds, **void** *fcn(**double *,double ***), **const char** *inp, **const char** *out, **Int_t** ns=100)
  - **tds**: an empty dataserver containing the input variables.
  - **fcn**: the function to analyse (a pointer).
  - **ns**: an integer to specify the number of simulations.
  - **out**: attribute names which represent the responses of the problem. it must refer to existing attributes in the "tds".
  - **inp**: input attribute names separated by colons. They represent all the inputs of the problem. They must refer to existing attributes in the "tds".
- `TRBD`(**`TDataServer`** *tds, **const char** *fcn, **Int_t** ns=100, **const char** *inp="", **const char** *out="")
  - **tds**: an empty dataserver containing the input variables.
  - **fcn**: the function to analyse (the function's name).
  - **ns**: an integer to specify the number of simulations.
  - **out**: attribute names which represent the responses of the problem. it must refer to existing attributes in the "tds".
  - **inp**: input attribute names separated by colons. They represent all the inputs of the problem. They must refer to existing attributes in the "tds".
- `TRBD`(**`TDataServer`** *tds, **`TCode`** *fcode, **Int_t** ns)
  - **tds**: an empty dataserver containing the input variables.
  - **fcode**: the code to analyse .
  - **ns**: an integer to specify the number of simulations.
- `TRBD`(**`TDataServer`** *tds, **`TRun`** *frun, **Int_t** ns)
  - **tds**: an empty dataserver containing the input variables.
  - **frun**: the runner to use code/function to analyse .
  - **ns**: an integer to specify the number of simulations.
- `generateSample`(**Option_t** *option=""): generate the sample used for computing the sensitivity indices.
  - **option**: no option has been implemented
- `computeIndexes`(**Option_t** *option=""): compute the Sobol sensitivity indices.
  - **option**: string specifying the drawIndexes option used .
- void `drawIndexes`(**`TString`** sTitre, **const char*** select="", **Option_t** *option=""): draw the coefficients.
  - **sTitre**: a string containing the title of the figure.
  - **select**: a string containing a solution.
  - **option**: a string containing the options of the graphics separated by commas.

VI.7 The Johnson relative weight

This section introduces indices whose purpose is mainly to obtain good estimators of the Shapley's values defined in [30]. The underlying assumption is to state that the model can be considered linear so that the results can be considered as proper estimation of the Shapley indices (with or without correlation between the input variables).

VI.7.1 General overview

In Uranie, computing sensitivity indices with Johnson's relative weights method is dealt with the eponymous class `TJohnsonRW` which inherits from the **`TSensitivity`** class.

This class handles all the steps to compute the indices depending on provided informations:

- one can only provide a correlation matrix without data. The indices would be estimated as such.
- one can provide a sample containing input and outputs variables. From there the class will compute the correlation matrix and estimate the indeces from them.
- If one provides either a code, a function, or a runner (see Section VIII.4) then, the class can
  - generate the deterministic sample if no data are found;
  - run the code, the analytic function or the evaluator if the output is not provided as well;
  - computing the indices.

VI.7.1.1 Example: simple computation of Johnson's relative wieght using a function

The example script below uses the `TJohnsonRW` class to compute and display the relative weights:

```
{
  gROOT->LoadMacro("UserFunctions.C");

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsflowrate", "DataBase flowrate");
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  // \param Size of a sampling.
  Int_t nS = 1000;
  TString FuncName="flowrateModel";

  TJohnsonRW *tjrw = new TJohnsonRW(tds, FuncName, nS, "rw:r:tu:tl:hu:hl:l:kw", FuncName);
  tjrw->computeIndexes();

  // Get the results on screen
  tjrw->getResultTuple()->Scan("Out:Inp:Method:Value","Order==\"First\"");

  // Get the results as plots
  TCanvas *cc = new TCanvas("canhist", "histgramme");
  tjrw->drawIndexes("Flowrate", "", "nonewcanv,hist,first");
  cc->Print("appliUranieFlowrateJohnsonRW1000Histogram.png");
  TCanvas *ccc = new TCanvas("canpie", "TPie");
  tjrw->drawIndexes("Flowrate", "", "nonewcanv,pie");
  ccc->Print("appliUranieFlowrateJohnsonRW1000Pie.png");

}
```

The figures resulting from this script are shown below and display the sensitivity indices in a histogram and in a pie chart. The structure of both plots and tables, as they are common with the rest of the sensitivity class, provides the results in the ntuple that can be reached with `getResultTuple()`. This explains why the results are stored in the *Sobol* column.

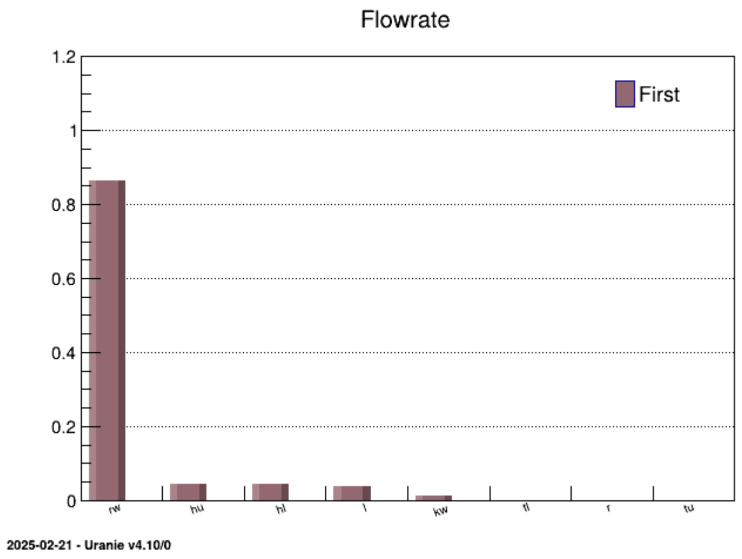


Figure VI.13: Histogram of JohnsonRW's indices

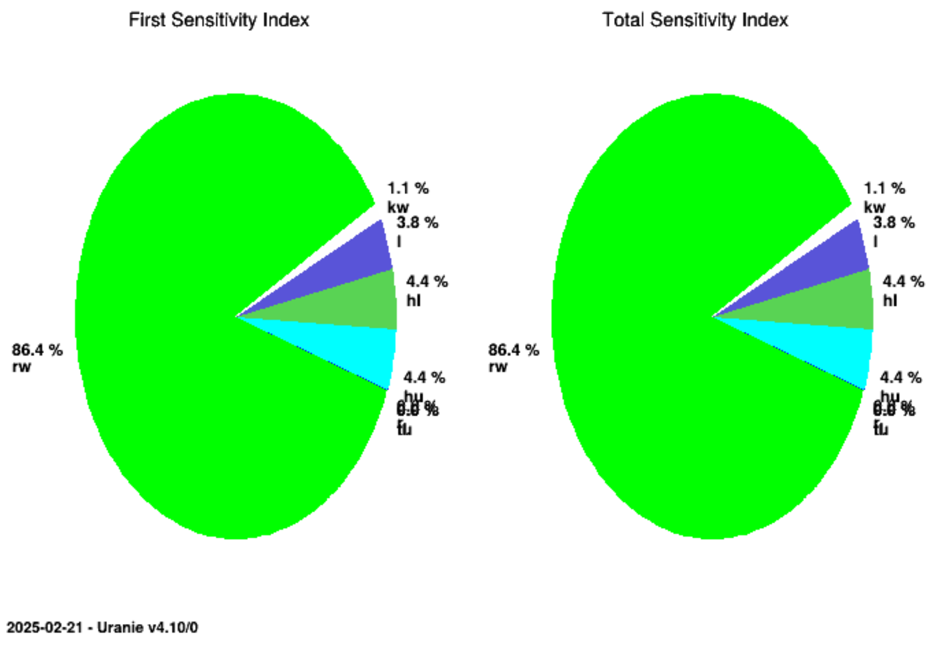


Figure VI.14: Pie chart of JohnsonRW's indices

VI.7.1.2 Specifying the input parameters

First, define the uncertain parameters and add them to a `TDataServer` object:

```
// Define the DataServer
TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

VI.7.1.3 `TJohnsonRW` constructor

There are four different kinds of constructors to build a **`TJohnsonRW`** object, each corresponding to a different problem:

- the model is either provided data or just a correlation matrix;
- the model is an analytic function run by Uranie,
- the model is a code run by Uranie.
- the model is either a function or a code and the problem is specified through a Relauncher architecture.

VI.7.1.3.1 `TJohnsonRW` constructor with provided data

The constructor prototype used when data are provide (or one will not used data but just a correlation matrix):

```
// Create a TJohnsonRW object data
TJohnsonRW(TDataServer *tds, const char *inp,  const char *out ,  const char *Option="")
```

This constructor takes up to four arguments:

- a pointer to a `TDataServer` object,
- a string to specify the names of the input factors separated by ’:’ (ex. "rw:r:tu:tl:hu:hl:l:kw"), no default is accepted,
- a string to specify the names of the output variables of the model, no default is accepted.
- a string to specify the options, by default empty "";

Here is an example of how to use the constructor with provided data:

```
TJohnsonRW * tjrw = new TJohnsonRW(tds, "rw:r:tu:tl:hu:hl:l:kw", "flowrateModel");
```

The way to inject the correlation matrix is discussed below.

VI.7.1.3.2 `TJohnsonRW` constructor for an analytic function

The constructor prototype used with an analytic function is:

```
// Create a TJohnsonRW object with an analytic function
TJohnsonRW(TDataServer *tds,  void *fcn,  const char *inp,  const char *out,  Int_t ns)
TJohnsonRW(TDataServer *tds,  const char *fcn,  Int_t ns,  const char *inp="",  const char ↩
    *out="")
```

This constructor takes five arguments:

- a pointer to a `TDataServer` object,
- a pointer to an analytic function (either a `void` or a `const char` that represents the function’s name when it has been loaded in ROOT’s memory),
- an integer to specify the number of simulations, its default value is 100,
- a string to specify the names of the input factors separated by ’:’ (ex. "rw:r:tu:tl:hu:hl:l:kw"), its default value might be the empty string "",
- a string to specify the names of the output variables of the model, its default value might be the empty string "".

Here is an example of how to use the constructor with an analytic function:

```
Int_t nS = 50;
TJohnsonRW * tjrw_func = new TJohnsonRW(tds, "flowrateModel", nS, "rw:r:tu:tl:hu:hl:l:kw",  ↩
    "flowrateModel");
```

VI.7.1.3.3 `TJohnsonRW` constructor for a code

The constructor prototype used with a code is:

```
// Create a TJohnsonRW object with a code
TJohnsonRW(TDataServer *tds, TCode *fcode,  Int_t ns, Option_t *option="")
```

This constructor takes three arguments:

- a pointer to a `TDataServer` object,
- a pointer to a **`TCode`**,
- an integer to specify the number of simulations.

Here is an example of the use of this constructor on the flowrate case:

```
// The reference input file
TString sJDDReference = TString("flowrate_input_with_keys.in");

// Set the reference input file and the key for each input attributes
tds->getAttribute("rw")->setFileKey(sJDDReference, "Rw");
tds->getAttribute("r")->setFileKey(sJDDReference, "R");
tds->getAttribute("tu")->setFileKey(sJDDReference, "Tu");
tds->getAttribute("tl")->setFileKey(sJDDReference, "Tl");
tds->getAttribute("hu")->setFileKey(sJDDReference, "Hu");
tds->getAttribute("hl")->setFileKey(sJDDReference, "Hl");
tds->getAttribute("l")->setFileKey(sJDDReference, "L");
tds->getAttribute("kw")->setFileKey(sJDDReference, "Kw");

// The output file of the code
TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
// The attribute in the output file
fout->addAttribute(new TAttribute("yhat"));

// Instanciation de mon code
TCode *mycode = new TCode(tds, "flowrate -s -k");
// Adding the outputfile to the tcode object
mycode->addOutputFile( fout );

// Size of a sampling.
Int_t nS2 = 50;
TJohnsonRW * tjrw2 = new TJohnsonRW(tds, mycode, nS2);
```

VI.7.1.3.4 `TJohnsonRW` constructor for a runner

The constructor prototype used with a runner is:

```
// Create a TJohnsonRW object with a runner
TJohnsonRW(TDataServer *tds, TRun *frun,  Int_t ns, Option_t *option="")
```

This constructor takes three arguments:

- a pointer to a `TDataServer` object,
- a pointer to a **`TRun`**,
- an integer to specify the number of simulations.

Here is an example of the use of this constructor on the flowrate code in sequential mode:

```
// The input file
TKeyScript infile("flowrate_input_with_keys.in");
// provide the input and their key
infile.setInputs(8, tds->getAttribute("rw"), "Rw", tds->getAttribute("r"), "R",
tds->getAttribute("tu"), "Tu", tds->getAttribute("tl"), "Tl", tds->getAttribute("hu"), "Hu" ↩
    ,
tds->getAttribute("hl"), "Hl", tds->getAttribute("l"), "L", tds->getAttribute("kw"), "Kw");

TAttribute yhat("yhat");
// The output file of the code
TKeyResult outfile("_output_flowrate_withKey_.dat");
// The attribute in the output file
outfile.addOutput(&yhat, "yhat");

// Instanciation de mon code
TCodeEval code("flowrate -s -k");
// Adding the intput/output file to the code
code.addInputFile(&infile);
code.addOutputFile(&outfile);

TSequentialRun run(&code);
run.startSlave();
if(run.onMaster())
{
    // Size of a sampling.
    Int_t nS2 = 50;
    TJohnsonRW * tjrwR = new TJohnsonRW(tds, &run, nS2);
    //...
}
```

VI.7.1.4 Not using a sample but only a correlation matrix

With the Johnson relative weight method, it is possible to use either data loaded from an existing file or no data at all but only corration matrix to estimate the weights and the R^2. This former is done, as usual, by calling the `fileDataRead` of the **TDataserver** object. This is not discussed in details as these as this aspect if fairly classical. The later on the other hand, is new and specific to the `TJohnsonRW` class. This possibility is very specific and can be used through the method `setCorrelationMatrix(TMatrixD Corr);`

Warning
It is important to pay attention not to mix up `setInputCorrelationMatrix`, `setCorrelationMatrix`. The former is useful to generate a correlated sample before submitting the computation with function or code (see Section VI.7.1.5 for instance) while the latter is a correlation matrix that does not only contains the input variables but also the output ones.

Here is an exemple of code that shows how to use this possibility. It starts with the input attribute definition (using `StochasticAttributes` is not compulsory anymore has no design-of-experiments will be drawn), along with the output ones. Once done, the full correlation matrix (inputs and outputs) can be provided through the `setCorrelationMat` (for more details on the `GenCorr` method, see Section XIV.5.18.2). The rest is pretty much straightforward.

```
// Load the GenCorr function
gROOT->LoadMacro("UserFunctions.C");

// Define the DataServer
```

```
TDataServer *tds = new TDataServer("tdsflowrate", "Ex. Flowrate");
tds->addAttribute("rw");
tds->addAttribute("r");
tds->addAttribute("tu");
tds->addAttribute("tl");
tds->addAttribute("hu");
tds->addAttribute("hl");
tds->addAttribute("l");
tds->addAttribute("kw");
// outputs
tds->addAttribute("flowrateModel");
tds->getAttribute("flowrateModel")->setOutput();

// Get the full correlation matrix
TMatrixD inCorr(8,8); //Input size definition
GenCorr(&inCorr, true, true);
// inCorr is now 9 by 9 as linearOutput has been added in GenCorr

TJohnsonRW * tjrw = new TJohnsonRW(tds, "rw:r:tu:tl:hu:hl:l:kw", "flowrateModel");
tjrw->setCorrelationMatrix(inCorr);
tjrw->computeIndexes();
```

VI.7.1.5 Generating the sample

To generate the sample that can be used to get the relative weights, one can use the `generateSample` method defined in `TSensitivity` but the purpose of the relative weight method is focusing on correlated inputs issue. In order to do this, one should specify the correlation structure of the inputs through the method `setInputCorrelationMatrix(TMat Corr)` where the only argument is a correlation matrix that has to be a symmetrical positive definite matrix whose coefficients have to be in $[-1,1]$ while the diagonal ones must be set to 1. With the usual variable definition, this could look like this:

```
TMatrixD inCorr(8,8);
double corrValue[64]={1,0.184641,-0.613412,-0.214481,0.373538,-0.0926293,0.656586,0.194991,
  0.184641,1,0.134385,0.0704593,-0.284958,0.105629,-0.536972,0.663751,
  -0.613412,0.134385,1,0.200747,-0.174041,0.0817216,-0.547402,0.0202687,
  -0.214481,0.0704593,0.200747,1,-0.128116,-0.248905,-0.263717,-0.0613039,
  0.373538,-0.284958,-0.174041,-0.128116,1,-0.0832623,0.397725,-0.675141,
  -0.0926293,0.105629,0.0817216,-0.248905,-0.0832623,1,0.111711,0.100442,
  0.656586,-0.536972,-0.547402,-0.263717,0.397725,0.111711,1,-0.302142,
  0.194991,0.663751,0.0202687,-0.0613039,-0.675141,0.100442,-0.302142,1};
inCorr.Use(8,8,corrValue);

//Create the jrw object
TJohnsonRW * tjrw = new TJohnsonRW(tds, "flowrateModel", nS, "rw:r:tu:tl:hu:hl:l:kw", " ↩
    flowrateModel");
tjrw->setInputCorrelationMatrix(inCorr);
tjrw->generateSample();
```

Then, the sample generated can be exported in a file and used outside of Uranie to compute the simulations associated.

VI.7.1.6 Computing the indices

To compute the indices, run the method `computeIndexes`:

```
tjrw->computeIndexes();
```

Note that this method is all inclusive: it constructs the sample, launches the simulations and computes the indices.

VI.7.1.7 Displaying the indices

To display graphically the coefficients, use the method `drawIndexes`:

```
void TSensitivity::drawIndexes(TString sTitre, const char *select, Option_t * option)
```

The method needs:

- a `TString` containing the title of the figure,
- a string containing a selection (empty if no selection),
- a string containing the options of the graphics separated by commas.

Some of the options available are:

- "nonewcanv": to not create a new canvas,
- "pie": to display a pie chart,
- "hist": to display a histogram,
- "first": to display the indices of the first order (with "hist" only),

In our example the use of this method is:

```
TCanvas *cc = new TCanvas("canhist", "histogramme");
tjrw->drawIndexes("Flowrate", "", "nonewcanv,hist,first");
cc->Print("appliUranieFlowrateJohnsonRW500Histogram.png");

TCanvas *ccc = new TCanvas("canpie", "TPie",5,64,1270,560);
tjrw->drawIndexes("Flowrate", "", "nonewcanv,pie");
ccc->Print("appliUranieFlowrateJohnsonRW500Pie.png");
```

VI.7.1.8 Extracting the coefficients

The coefficients, once computed, are stored in a `TTree`. To get this `TTree`, use the method `TSensitivity::getResul`

```
TTree * results = tjrw->getResultTuple();
```

Several methods exist in ROOT to extract data from a `TTree`, it is advised to look for them into the ROOT documentation. We propose two ways of extracting the value of each coefficient from the `TTree`.

VI.7.1.8.1 Method of extraction

The method use the method `getValue` of the `TJohnsonRW` object specifying the order of the extract value ("First"), the related input and possibly more selected options.

```
double hl_first_index = tjrw->getValue("First","hl");
```

VI.7.1.8.2 Second method of extraction

The second method uses 3 steps to extract an index:

- scan the `TTree` for the chosen input variable (with a selection) in order to obtain its row number. In our example, if we chose the variable "hl", we'll use the command:

  ```
  results->Scan("*","Inp==\"hl\"");
  ```

  and in the resulting figure below we see that the first order index of "hl" is in the row 10:

  ```
  ************************************************************************
  *    Row   *       Out *  Inp *  Order * Method *      Algo *     Value *
  ************************************************************************
  *       10 * flowrateM *   hl *  First * Johnso * --first-- * 0.0435594 *
  *       11 * flowrateM *   hl *  Total * Johnso * --total-- * 0.0435594 *
  ************************************************************************
  ```

- set the entry of the `TTree` on this row with the method `GetEntry`;
- get the value of the index with `GetValue` method on the "Value" leaf of the `TTree`.

Below is an example of extraction of the index for "hl" in our flowrate case:

```
results->Scan("*","Inp==\"hl\"");
results->GetEntry(100);
Double_t S_Rw_Indexe =  results->GetLeaf("Value")->GetValue();
```

VI.7.1.8.3 Third method of extraction

The third method uses 2 steps to extract an index:

- use the `Draw` method with a selection to select the index, for example the selection for the first order index of "rw" is "Inp==\"rw\" && Algo==\"--first--\"";
- get the pointer on the value of the index with `GetV1` method on the `TTree`.

Below is another example of extraction of the first order index for "rw" in our flowrate case:

```
results->Draw("Value","Inp==\"rw\" && Algo==\"--first--\"","goff");
Double_t S_Rw_IndexeS = results->GetV1()[0];
```

Summary: `TJohnsonRW` constructors

- `TJohnsonRW`(**`TDataServer`** *tds, **const char** *inp, **const char** *out, **Option_t** *option="")
  - **tds**: an empty dataserver containing the input variables or a dataserver filled with an already existing design-of-experiments (input and output data available).
  - **out**: attribute names which represent the responses of the problem. it must refer to existing attributes in the "tds".
  - **inp**: input attribute names separated by colons. They represent all the inputs of the problem. They must refer to existing attributes in the "tds".
  - **option**: usual option field.
- `TJohnsonRW`(**`TDataServer`** *tds, **void** *fcn(**double** *,**double** *), **const char** *inp, **const char** *out, **Int_t** ns=100)
  - **tds**: an empty dataserver containing the input variables.
  - **fcn**: the function to analyse (a pointer).
  - **ns**: an integer to specify the number of simulations.
  - **out**: attribute names which represent the responses of the problem. it must refer to existing attributes in the "tds".
  - **inp**: input attribute names separated by colons. They represent all the inputs of the problem. They must refer to existing attributes in the "tds".
- `TJohnsonRW`(**`TDataServer`** *tds, **const char** *fcn, **Int_t** ns=100, **const char** *inp="", **const char** *out="")
  - **tds**: an empty dataserver containing the input variables.
  - **fcn**: the function to analyse (the function's name).
  - **ns**: an integer to specify the number of simulations.
  - **out**: attribute names which represent the responses of the problem. it must refer to existing attributes in the "tds".
  - **inp**: input attribute names separated by colons. They represent all the inputs of the problem. They must refer to existing attributes in the "tds".
- `TJohnsonRW`(**`TDataServer`** *tds, **`TCode`** *fcode, **Int_t** ns)
  - **tds**: an empty dataserver containing the input variables.
  - **fcode**: the code to analyse .
  - **ns**: an integer to specify the number of simulations.
- `TJohnsonRW`(**`TDataServer`** *tds, **`TRun`** *frun, **Int_t** ns)
  - **tds**: an empty dataserver containing the input variables.
  - **frun**: the runner to use code/function to analyse .
  - **ns**: an integer to specify the number of simulations.

Summary: `TJohnsonRW` methods

- `setInputCorrelationMatrix`(**TMatrixD** inCorr): define the correlation matrix that is used to intricate the input variables one to another.
  - **inCorr**: a $n_X \times n_X$ symmetrical positive definite matrix whose coefficients should all be in $[-1,1]$ but the diagonal ones which should be set to 1.
- `setCorrelationMatrix`(**TMatrixD** Corr): define the global correlation matrix that is used to intricate the input and output variables one to another.
  - **Corr**: a $(n_X+n_Y)\times(n_X+n_Y)$ symmetrical positive definite matrix whose coefficients should all be in $[-1,1]$ but the diagonal ones which should be set to 1.
- `generateSample`(**Option_t** *option=""): generate the sample used for computing the sensitivity indices.
  - **option**: no option has been implemented
- `computeIndexes`(**Option_t** *option=""): compute the Sobol sensitivity indices.
  - **option**: string specifying the drawIndexes option used .
- void `drawIndexes`(**TString** sTitre, **const char*** select="", **Option_t** *option=""): draw the coefficients.
  - **sTitre**: a string containing the title of the figure.
  - **select**: a string containing a solution.
  - **option**: a string containing the options of the graphics separated by commas.

VI.8 Sensitivity Indices based on HSIC

VI.8.1 Introduction to sensitivity measures using HSIC

This section introduces sensivity measures based on Hilbert-Schmidt independence criterion (HSIC)

VI.8.1.1 Example: simple computation of HSIC measures using a function

The example script below uses the `THSIC` class to compute and display the indices. Note that HSIC indexes do not require a specific design of experiments, they can be computed on a given sample.

```
void sensitivityHSICFunctionFlowrate(){

gROOT->LoadMacro("UserFunctions.C");

// Define the DataServer
TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

// Generation of the sample (it can be a given sample).
Int_t nS = 500;
TSampling *sampling = new TSampling(tds, "lhs", nS);
sampling->generateSample();

TLauncherFunction * tlf = new TLauncherFunction(tds, "flowrateModel");
tlf->setDrawProgressBar(kFALSE);
tlf->run();

// Create a THSIC object, compute indexes and print results
THSIC * thsic = new THSIC(tds, "rw:r:tu:tl:hu:hl:l:kw","flowrateModel");
thsic->computeIndexes("quiet");
thsic->getResultTuple()->SetScanField(60);
thsic->getResultTuple()->Scan("Out:Inp:Method:Order:Value:CILower:CIUpper");

// Print HSIC indexes
TCanvas  *can = new TCanvas("c1", "Graph for the Macro sensitivityHSICFunctionFlowrate" ↩
    ,5,64,1270,667);
thsic->drawIndexes("Flowrate", "", "hist,first,nonewcanv");

}
```

VI.8.1.2 **THSIC** constructor

There is one kind of constructor to build a **THSIC** object based on provided data. The HSIC compute indices on given data, the `TDataServer` must be filled. The constructor does not require a model to run (this has to be done).

```
// Create a THSIC object data
THSIC(TDataServer *tds, const char *inp,  const char *out ,  const char *Option="")
```

This constructor takes up to four arguments:

- a pointer to a `TDataServer` object,
- a string to specify the names of the input factors separated by ':' (ex. "rw:r:tu:tl:hu:hl:l:kw"), no default is accepted,
- a string to specify the names of the output variables of the model, no default is accepted.
- a string to specify the options, by default empty "". Possibility are "empiri" or "median" to specify the method to estimate the variance (see Section VI.8.1.4).

Here is an example of how to use the constructor with provided data:

```
THSIC * thsic = new THSIC(tds, "rw:r:tu:tl:hu:hl:l:kw", "flowrateModel");
```

VI.8.1.3 Computing the indices

To compute the indices, run the method `computeIndexes`:

```
thsic->computeIndexes();
```

Note this method computes the 'V-stat' indices. If the "unbiased" option is provided, it computes the 'U-stat' indices. It computes three values for each variable:

- the HSIC measures,
- the R2HSIC indices, which are considered as the sensitivity indices
- the p-value of the independence test

VI.8.1.4 Set the parameters of the gaussian kernel

`setEstimatedVariance` : Set the method used to estimate the variance of the gaussian kernel (`kUnknown` | `kStdEmpirical` | `kMedianeDelta`)

- `kStdEmpirical` : the parameters is estimated by the emperical standatd deviation of the attributes
- `kMedianeDelta` : use the median distance between points

```
thsic->setEstimatedVariance(URANIE::Sensitivity::THSIC::kStdEmpirical);
```

VI.8.1.5 Computing the p-value

Two technics are available to compute the p-values to test the independence between inputs and outputs:

- By default, the gamma approximation is used.
- The following command allows using the bootstrap

```
thsic->computeIndexes("nperm=100"); // specify the number of permutation
```

- `computeIndexes("nperm=200")` : Specify the number of bootstrap permutation using the option "nperm=" in `computeIndexes`

VI.8.1.6 Extracting the measure

The measures, once computed, are stored in a `TTree`. To get this `TTree`, use the method `TSensitivity::getResultTupl`

```
TTree * resultsHSIC = thsic->getResultTuple();
```

Several methods exist in ROOT to extract data from a `TTree`, it is advised to look for them into the ROOT documentation. We propose two ways of extracting the value of each coefficient from the `TTree`.

VI.8.1.6.1 Method of extraction

The method use the method `getValue` of the `THSIC` object specifying the order of the extract value ("R2HSic", "HSic" or "pValues"), the related input and possibly more selected options.

```
double hl_r2hsic_index = thsic->getValue("R2HSic","hl");
double hl_hsic_index = thsic->getValue("HSic","hl");
double hl_pvalues_index = thsic->getValue("pValues","hl");
```

Summary: THSIC constructors

- THSIC(**TDataServer** *tds, **const char** *inp, **const char** *out, **Option_t** *option="")
  - **tds**: a dataserver filled with an already existing design-of-experiments (input and output data available).
  - **out**: attribute names which represent the responses of the problem. it must refer to existing attributes in the "tds".
  - **inp**: input attribute names separated by colons. They represent all the inputs of the problem. They must refer to existing attributes in the "tds".
  - **option**: usual option field.

Summary: THSIC methods

- computeIndexes(**Option_t** *option=""): compute the HSIC sensitivity indices.
  - **option**: string specifying the drawIndexes option used .
- void drawIndexes(**TString** sTitre, **const char*** select="", **Option_t** *option=""): draw the coefficients.
  - **sTitre**: a string containing the title of the figure.
  - **select**: a string containing a solution.
  - **option**: a string containing the options of the graphics separated by commas.

Chapter VII

The Optimizer module

VII.1 Introduction

In the Optimisation module of Uranie, the word *optimisation* has two meanings:

1. The search for the optimum set of *values* of a function that takes its values in $\mathbb{R}$. This optimisation research regroups two realities:

   - the search for the set of inputs that produce the optimum of a function (analytic function or a computational code);
   - the search for the optimal set of parameters of a model which minimises the *difference* between reference values and estimations from the model got from computational code. This *difference* may be the root mean squared error or a weighted root mean squared error.

2. The search for a set of inputs that minimise a cost function that takes its values in $\mathbb{R}^p$ $(p > 1)$, *i.e.* multi-objective optimisation, what represents the search of the Pareto frontier in the costs space.

Tip
In order to avoid confusion, please note that the word **function** will be used to name either a **C++ function** or an **external code**.

In this section, when dealing about an optimisation, we will always refer to *minimisation*, keeping in mind that the maximisation of a function is equivalent to the minimisation of its opposite. A discussion on the general concept of optimisation (with the difference between global and local and an simple introduction to the Pareto frontier, can be found in [30]). The multi-criteria discussion has been moved to the Reoptimizer part (in Chapter IX). The use several multi-criteria algorithms, gathered in the Vizir package, is now only supported with the implementation done in the Reoptimizer package.

VII.2 Function optimisation

The first optimisation type is the search for the optimum set of *values* of a function that takes its values in $\mathbb{R}$.

VII.2.1 Rosenbrock function

To illustrate the usage of the Optimizer module with (both analytic and computational) functions, we are going to use the Rosenbrock function from the literature [42].

VII.2.1.1 Presentation of the problem

The Rosenbrock function is widely used in optimisation libraries. It is a function of $\mathbb{R}^2$ in $\mathbb{R}$. It is defined, for two parameters {a, b} in $\mathbb{R}$, by:

$$f(x_1,x_2) = a \times (x_2 - x_1^2)^2 + b \times (1.0 - x_1)^2$$

EQUATION VII.1: Rosenbrock function

In the illustration below, the two parameters are set to: a = 10.0 and b = 1.0.

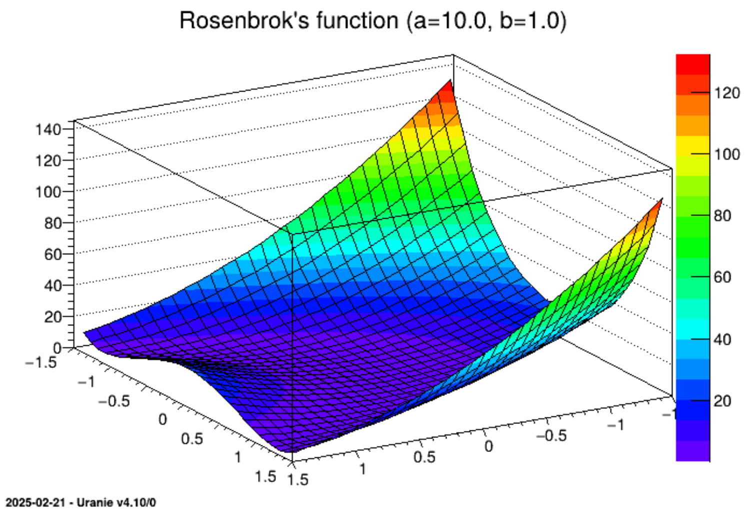


Figure VII.1: 3D representation of the Rosenbrock function

By analytic computation, it can be proved that the Rosenbrock function is minimum in (1, 1) for all the values of a and b, if parameters a and b are both positive.

VII.2.1.2 Case of an analytic function

The Rosenbrock function is defined in the file `UserFunctions.C` (in the macros folder `${URANIESYS}/share/uranie/ma`
with the prototype:

```
void rosenbrock(Double_t *x, Double_t *y)
```

Its content is as follow:

Uranie source code of the analytic function "rosenbrock"

```
void rosenbrock(Double_t *x, Double_t *y)
{
Double_t dx1 = x[0], dx2 = x[1];
Double_t da=10.0, db=1.0;

y[0] = da * (dx2-dx1*dx1)**2+db*(1.0-dx1)**2;
}
```

VII.2.1.3 Case of an external code

Also provided with Uranie comes an executable version of the Rosenbrock function. In this case, the Rosenbrock function is not an analytic function as described above anymore, but a computational code compiled with Uranie, that deals with input and output files.

VII.2.1.3.1 Usage of the code

Below are presented the available parameters of the rosenbrock executable (available in the `${URANIESYS}/bin` folder):

```
Usage: rosenbrock [-d] [-v] [-k|-r] [file]
-v:   verbose mode
-d:   debug mode (prints most of its steps in order to debug the code in case of problem)
-k:   input file with keys  [default is input_rosenbrock_with_keys.dat]
-r:   input file with only values in rows [default is input_rosenbrock_with_values_rows.dat ↩
    ]
file:   name of the input file, if different from the default name
```

- **-v** option allows to print some messages to know the state of the code
- **-d** option allows to give some intermediate values to users while executing **rosenbrock**, such as values read in input files, or to show steps or information in execution.
- **-k** options allow to simulate the launch of **rosenbrock** with data files of "key=value" format. If user does not specify an input file, **rosenbrock** will look for the input file *input_rosenbrock_with_keys.dat* whom "key=value" format is described below.
- **-r** option allows to simulate the launch of **rosenbrock** code with files with values in rows. The default input file is *input_rosenbrock_with_values_rows.dat* whom "values in rows" format is described below.

Note that the input file must contain at least two values, for the parameters x and y. Values for a and b can be omitted, in which case default values 10 and 1 will be respectively taken.

VII.2.1.3.2 Input files

In this section are described the file formats the rosenbrock executable can read.

- When used with the "-k" option, the rosenbrock executable takes its parameters from input files with format "key=value", where the keys are the variable names, *i.e.* x, y, a and b. By default, the file *input_rosenbrock_with_keys.dat* is taken as an input file, but if the user creates a new file, it must keep the same format. Below is the content of the default file, *input_rosenbrock_with_keys.dat*:

```
#
#
# inputfile for the \b rosenbrock code
# \date   mar jui 3 14:38:43 2007
# the two parameters
#

x  = -1.20 ;
y  = 1.0 ;
a = 10.0 ;
b = 1.0 ;
```

As presented below, if the user does not want to specify the values for parameters a and b, the following file will produce the same result:

```
x  = -1.20 ;
y  = 1.0 ;
```

- When used with the "-r" option, the rosenbrock executable gets the parameters from input files where values are stored in a single row, in the following order: x, y, a and b. The default file is *input_rosenbrock_with_values_rows.dat*, which is taken if no file is specified. Below is the content of the default file, *input_rosenbrock_with_values_rows.dat*:

```
-1.20 1.0
```

VII.2.1.3.3 Output files

The `rosenbrock` code generates two output files, in which are stored both input attributes and output values:

- *_output_rosenbrock_with_values_rows_.dat*: file with "row" format and with a header containing the names of the variables, and then their values:

```
#COLUMN_NAMES: x | y | fval | pA | pB

-1.200000e+00   1.000000e+00    6.776000e+00    1.000000e+01    1.000000e+00
```

- *_output_rosenbrock_with_keys_.dat*: file with "key=value" format without any header:

```
X = -1.200000e+00 ;
Y = 1.000000e+00 ;
fval = 6.776000e+00 ;
fA = 1.000000e+01 ;
fB = 1.000000e+00 ;
```

VII.2.2 `TOptimizer` constructors

Function optimisation is dealt in Uranie with the `TOptimizer` class. In order to manage both types of evaluation functions (C++ functions or external codes), two constructors are available.

VII.2.2.1 Constructor for C++ function optimisation

To build a `TOptimizer` object from a C++ function, use the following constructor:

```
TOptimizer(TDataServer *tds, void (*fcn)(double*,double*), TString sinput, TString soutput)
TOptimizer(TDataServer *tds, const char *fcn, TString sinput="", TString soutput="")
```

This is used to create a TOptimizer object from the TDataServer object, and an analytic function either given by a void pointer (first line) or by its name (second line) when it has been loaded in ROOT's memory. The rest of the arguments are the list of the input attributes found in the TDataServer, sinput, and the output attribute to be created in the TDataServer, soutput. These two arguments are now compulsory when using a constructor with a pointer of function (for more details on this, see Section I.2.5).

Note that the parameter sinput can be left empty, and the value "" is then interpreted as "all the input attributes".

VII.2.2.2 Constructor for external code optimisation

To build a TOptimizer object from an external code, use the following constructor:

```
TOptimizer( TDataServer  *tds, TCode *code)
```

This is used to create a TOptimizer object from the TDataServer, and the TCode external code.

Tip
To find information about the creation of a TCode, please refer to the Section IV.3.

VII.2.3 Optimisation as minimum of function seeking

The first goal of the optimisation module of Uranie is to find the minimum of a function (C++ function or external code), defined from $\mathbb{R}^n$ in $\mathbb{R}$.

VII.2.3.1 Selecting the algorithm

Two algorithms can be used to seek the minimum of a function: the Migrad or Simplex algorithms, included in the Minuit package of ROOT. Below are excerpts from the Which Minimizer to Use page of Minuit2, ROOT's optimiser library that can be used in Uranie:

Migrad algorithm

This is the best minimizer for nearly all functions. It is a variable-metric method with inexact line search, a stable metric updating scheme, and checks for positive-definiteness. [...] Its main weakness is that it depends heavily on knowledge of the first derivatives, and fails miserably if they are very inaccurate.

Simplex algorithm

This genuine multidimensional minimisation routine is usually much slower than MIGRAD, but it does not use first derivatives, so it should not be so sensitive to the precision of the function calculations, and is even rather robust with respect to gross fluctuations in the function value. However, it gives no reliable information about parameter errors, no information whatsoever about parameter correlations, and worst of all cannot be expected to converge accurately to the minimum in a finite time.

In order to select the algorithm to be used, apply the method `setMethod` to the `TOptimizer` object, with argument one of the keywords `kMigrad` or `kSimplex` respectively for Migrad or Simplex algorithm:

```
TOptimizer * topt = new TOptimizer(tds, myCode);
topt->setMethod(TOptimizer::kSimplex);
// or opt->setMethod(TOptimizer::kMigrad);
```

If not specified, the default algorithm used by the Optimizer module is the Migrad algorithm.

VII.2.3.2 Selecting the cost attribute

As was previously mentioned, the `TOptimizer` class is dedicated to the research of the minimum of functions that take values in $\mathbb{R}$. But it is also possible to find the minimum of a function that returns multiple values that we want to minimise along one of these values, by selecting the output value to consider as the one to minimise. To select the return value the function will be optimised for, use the `selectCost` method, with the name of the output among which the optimisation will be performed as argument:

```
// definition of the output file
TOutputFileDataServer *fOutputFile = new TOutputFileDataServer(" ↩
    _output_rosenbrock_with_values_rows_.dat");
fOutputFile->addAttribute(new TAttribute("fa"));
fOutputFile->addAttribute(new TAttribute("fb"));
fOutputFile->addAttribute(new TAttribute("fval"));

// definition of the code
TCode *mycode = new TCode(tds,"rosenbrock -r");
mycode->addOutputFile( fOutputFile );

// TOptimizer
TOptimizer * toptC = new TOptimizer(tds, mycode);
toptC->selectCost("fval");  // "fval" is the chosen cost for the optimisation
```

VII.2.3.3 Setting parameters to the algorithm

In order to improve the quality of the optimisation, or to control the time the computation will last, the user can modify a few parameters:

- `setTolerance(Double_t dtol)` sets the tolerance of the optimisation process to `dtol`. Default value is 0.01.
- `setMaxIterations(Int_t nmax)` sets the maximum number of iterations of the optimisation process to `nmax`. Note that the `TDataServer` is saved after each iteration. Default value is 50.
- `setMaxFunctionCalls(Int_t nmax)` sets the maximum number of calls to the function to minimise to `nmax`. Default value is 10.000.

Tip
Note that at each iteration, several function calls may be performed.

VII.2.3.4 Adding variables on the fly

When dealing with an external code, which is not easily accessible, new possibilities have been introduced to allow to add both input and output variables defined with simple mathematical formulaes. The examples discussed below are taken from the two use-cases, provided in Section XIV.7.2 and Section XIV.7.4 respectively for a function and an external code.

New input variables can be created with the `addAttribute` method that takes a name and a formula as arguments.

```
tds->addAttribute("xshift","x-0.1");
tds->addAttribute("yshift","y+0.2");
```

Once done, the `TOptimizer` should be warned to used these variables by:

- specifying the list of inputs for a function constructor:

```
gROOT->LoadMacro("UserFunctions.C");
TOptimizer *toptfunc = new TOptimizer(tds, "rosenbrock", "xshift:yshift","out");
```

- setting the way to write these inputs in the input file (instead of the original ones)

```
tds->getAttribute("xshift")->setFileKey("input_rosenbrock_with_keys.dat","x");
tds->getAttribute("yshift")->setFileKey("input_rosenbrock_with_keys.dat","y");
```

New output variables are introduced by calling a new method called `addOutputVariable` whose only argument is the list of formulaes to be applied separated by semi-colons. In both cases (function and code) it looks like this:

```
topt->addOutputVariable("fval+1:fval*fval:fval*3");
```

Once done, the optimisation can be done on any of these newly defined variable, simply using the `selectCost` as

```
topt->selectCost("fval+1");
```

Warning
These functionnalities are tested both for code and function optimisation, but we do not recommend to use it in the latter case. These computations are indeed requesting `TTree` operations which are slow with respect to the execution's time of an analytic function. In this case we strongly recommend to modified the function at hand to get the expected variables.

VII.2.4 Optimisation as code adjustment

The second goal of the optimisation module of Uranie is to find the optimal set of parameters of a model which minimises the *distance* between reference values and estimation from the model.

VII.2.4.1 Creation of objectives

Two *distances* are implemented:

- The root mean square deviation:

$$\text{obj} = \frac{\alpha}{n_S}\sum_{i=1}^{n_S}(y_i^\star - \hat{y}_i)^2$$

where:

- $y^\star$ is the reference values vector;
- $\hat{y}$ is the estimated values vector;
- α is a weight coefficient for the objective.

This distance is used with the following `TOptimizer` method:

```
addObjective(TString name,
TDataServer *tds,
TString ystar,
TOutputFile *outfile,
TString yhat,
Double_t weight)
```

The parameters are:

- **name**: the name of the objective to add;
- **tds**: the `TDataServer` that contains the reference values;
- **ystar**: the name of the output attribute in the `TDataServer` tds;
- **outfile**: the `TOutputFile` where the output values of the code are stored;
- **yhat**: the name of the output attribute in the output file;
- **weight** (optional): a coefficient to multiply the result by. The default value, if not specified, is 1.

• the weighted root mean square deviation:

$$\text{obj} = \alpha \sum_{i=1}^{ns} \frac{(y_i^\star - \hat{y}_i)^2}{\sigma_i^2}$$

where:

- $y^\star$ is the reference values vector;
- $\hat{y}$ is the estimated values vector;
- σ is a weight coefficients vector for the different values;
- α is a weight coefficient for the objective.

This distance is used with the following `TOptimizer` method:

```
addObjective(TString name,
TDataServer *tds,
TString ystar,
TString sigma,
TOutputFile *outfile,
TString yhat,
Double_t weight)
```

The parameters are:

- **name**: the name of the objective to add;
- **tds**: the `TDataServer` that contains the reference values;
- **ystar**: the name of the output attribute in the `TDataServer` tds;
- **sigma**: the name of the weight attribute in the `TDataServer` tds (used only in the second method);
- **outfile**: the `TOutputFile` where the output values of the code are stored;
- **yhat**: the name of the output attribute in the output file;
- **weight** (optional): a coefficient to multiply the result by; the default value if not specified is 1.

VII.2.4.2 Manipulation of the objectives

The objectives previously defined can be set active or inactive thanks to the methods `activeObjective(TString name)` and `unactiveObjective(TString name)`, which both take the name of the objective as a parameter.

In order to easily invert the state of all the objectives defined, use the `invertObjective()` method. By default, all the objectives added are actives.

VII.2.4.3 Manipulation of the parameters

In order to consider attributes as fixed parameters, they can be set fixed or unfixed thanks to the methods `fixParameter(T name)` and `unfixParameter(TString name)`, which both take the name of the objective as a parameter.

In order to easily invert the state of all the parameters defined, use the `invertParameters()` method. By default all the parameters are unfixed.

VII.2.5 Performing the optimisation

Once the optimisation object is constructed and associated with costs/objectives, the optimisation is simply performed by calling the `optimize()` method. The call to this method fills the dataserver specified in the constructor with the best solution found by the optimisation algorithm.

Summary: TOptimizer

- Constructors:
  - `TOptimizer`(**TDataServer** *tds, **void/const char** *fcn, **TString** sinput="", **TString** soutput="")
    Creates a `TOptimizer` object from the `TDataServer`, and an analytic function (given either by a `void` or a `const char` representing the function's name when loaded in ROOT's memory). The other arguments are optional and are the input and output attributes.
  - `TOptimizer`(**TDataServer** *tds, **TCode** *code)
    Creates a `TOptimizer` object from the `TDataServer`, and the `TCode` external code.

- For minimum of function seeking:
  - `setMethod`(**EOptimMethod** method=kMigrad): Sets the minimum seeking method to `method` (kMigrad or kSimplex).
  - `selectCost`(**TString** scost): Sets the cost attribute to `scost`.
  - `addOutputVariable`(**TString** soutputs): Adds new output by defining formulaes based on existing variables. These newly defined variables can be used as cost function by calling the `selectCost` method.
  - `setTolerance`(**Double_t** dtol): Sets the tolerance of the optimisation process. Default value is 0.01.
  - `setMaxIterations`(**Int_t** nmax): Sets the maximum number of optimisation iterations. Default value is 50.
  - `setMaxFunctionCalls`(**Int_t** nmax): Sets the maximum number of calls to the function to minimise. Default value is 10000.

- For code adjustment:
  - `addObjective`(**TString** name, **TDataServer** *tds, **TString** ystar, **TOutputFile** *outfile, **TString** yhat, **Double_t** weight): Adds the objective:

$$\mathrm{obj} = \frac{\alpha}{ns}\sum_{i=1}^{ns}(y_i^\star - \hat{y}_i)^2$$

  - `addObjective`(**TString** name, **TDataServer** *tds, **TString** ystar, **TString** sigma, **TOutputFile** *outfile, **TString** yhat, **Double_t** weight): Adds the objective:

$$\mathrm{obj} = \alpha\sum_{i=1}^{ns}\frac{(y_i^\star - \hat{y}_i)^2}{\sigma_i^2}$$

  - `activeObjective/unactiveObjective`(**TString** name): Active/unactive the objective `name`.
  - `invertObjective`(): Invert the activation of all the objectives.
  - `fixParameter/unfixParameter`(**TString** name): Fix/unfix the parameter `name`.
  - `invertParameters`(): Invert the fix status of all the parameters.

- For launching the optimisation process:
  - `Optimize(TString="same")`: Performs the optimisation and fills the tds specified in the constructor

VII.3 Multicriteria optimisation

The implementation of the Vizir algorithms within the Optimizer module being considered deprecated, this part is now discussed in Chapter IX.

Chapter VIII

The Relauncher module

VIII.1 Introduction

The aim of the Relauncher module is to provide a general architecture for all parametric study and is, because of this generality aspect, heavily used throughout the Uranie platform. However, it is generally used for more advanced techniques than the usual recommended first steps and it allows more flexible distribution approaches.

Studies allowed thanks to this module (no concrete study will indeed be described in this chapter, as the module is more a support for many other classes in other modules) aim at evaluate a model for different input parameter's values and check the evolution of its outputs. These studies can be split into two kinds:

- the opened-loop ones: all input parameters are known at the start (Monte-Carlo simulation for instance).
- the closed-loop ones: results of the evaluation will impact the next input parameter's value (optimisation for instance).

One can find examples of how to run analysis with the relauncher implementation in Section XIV.8.

Item evaluations can be time consuming, and many kinds of such studies are able to distribute them on computer resources. This architecture provides different ways to use these resources. However, if evaluation is fast, evaluation distribution is counterproductive.

Because of the very specific organisation of this module, the class hierarchy is not shown here but split into pieces which will be introduced in the next section. Every component is discussed in more details in the following sections and a schematic description of the needed steps to define a relauncher procedure is shown in Figure VIII.1.

VIII.2 Relauncher abstraction levels

In order to obtain a modular system, three abstraction levels are distinguished:

- The top level (`TMaster`) deals with the problem, and defines items to be evaluated. It plays a role of supervisor and this level is usually called *master*.
- The intermediate level (`TRun`) defines where evaluations are computed using available computer resources. This level is usually called *runner*.
- The bottom level (`TEval`) uses functions provided by the user to evaluate item characteristics and this level is usually called *assessor*.

Ideally, the combination of any classes of this three levels are possible. In real world, some combinations are useless (do not distribute a sequential algorithm) or impossible (many algorithm cannot converge if an non-calculable evaluation occurs).

As you can see from the defined layers, the architecture is designed to realise many evaluations in parallel. It deals both with a parallel evaluation and with a parallel studies (island optimisation for example). A feasibility test have been done with a parallel evaluation, but it needs extra works from the user.

Now let's start with a first *hello world* program. It treats the classical Rosenbrock optimisation problem (already introduced in Section VII.2.1). The script below starts with namespace directive, continues with the user supplied evaluation function, and ends with the study procedure. This procedure follows a bottom-up definition: used variables, evaluation function declaration, used resources, and study. Variables are used to define the evaluation prototype and the study (item definition variables, etc), and link these definitions together. In this example, some declarations may seem redundant, but they show their relevance in a more complicated example.

```
using namespace URANIE::DataServer;
using namespace URANIE::Relauncher;
using namespace URANIE::Reoptimizer;

void rosenbrock(double *in, double *out)
{
    double x, y, d1, d2;
    x = in[0]; y = in[1];
    d1 = (1-x);
    d2 = (y-x*x);
    out[0] = d1*d1 + d2*d2;
}

void Rosenbrock()
{
    // problem variables
    TAttribute x("x", -3.0, 3.0),
               y("y", -4., 6.),
               ros("rose");

    // user evaluation function
    TCIntEval eval(rosenbrock);
    eval.addInput(&x);
    eval.addInput(&y);
    eval.addOutput(&ros);

    // resources
    TSequentialRun run(&eval);
    run.startSlave();
    if (run.onMaster()) {
        // data server
        TDataServer tds("rosopt", "Rosenbrock Optimisation");
        tds.addAttribute(&x);
        tds.addAttribute(&y);

        // optimisation
        TVizirGenetic algo;
        TVizir2 study(&tds, &run, &algo);
        study.addObjective(&ros);
        study.solverLoop();

        // save results
        tds.exportData("rosenbrock.dat");

        run.stopSlave();
    }
}
```

This example will not be detailed, it is showed for its structure that you may find again on other studies. however, it must be self explanatory.

An important aspect needs to be pointed out. It concerns the resource handling. This script deals with both the study side and the evaluation side which may be treated by different resources. Both sides need to know variables and evaluation function, while study objects are only useful on study side. Here is the frame of the code that deals with it.

```
void Study()
{
// both side definitions
...

run.startSlave(); // slave is evaluation side
if (run.onMaster()) { // master is study side
// study definitions
...

run.stopSlave();
}
}
```

A translation of this code may be: once one starts the study, evaluation-needed objects are defined and evaluation-resources could start their loop waiting for items. The study-resource are allocated from the `onMaster` method and many things are done from there: distributing the evaluations and collecting the results. Once the study is finished (or at least, no more evaluation is needed), evaluation-resources can be stopped: they stop their loop, exit and jump the study instruction bloc which is no concern to them.

Finally the use of **Relauncher** module can be sketched in a four-steps process starting as usual, by defining the problem/model to be tested, defining the rule to be applied on the various inputs (meaning configuring the assessors also knowing the way one wants to run these calculations), choosing the corresponding runner and launch the computation through the `TLauncher2` instantiation (or any other *TMaster* inheriting class, such as the ones defined in the Reoptimizer module for optimisation problem, see Chapter IX). The colored arrows in Figure VIII.1 show the allowed associations of assessors and runners (to respect, for instance, the thread-safe properties of the assessors).

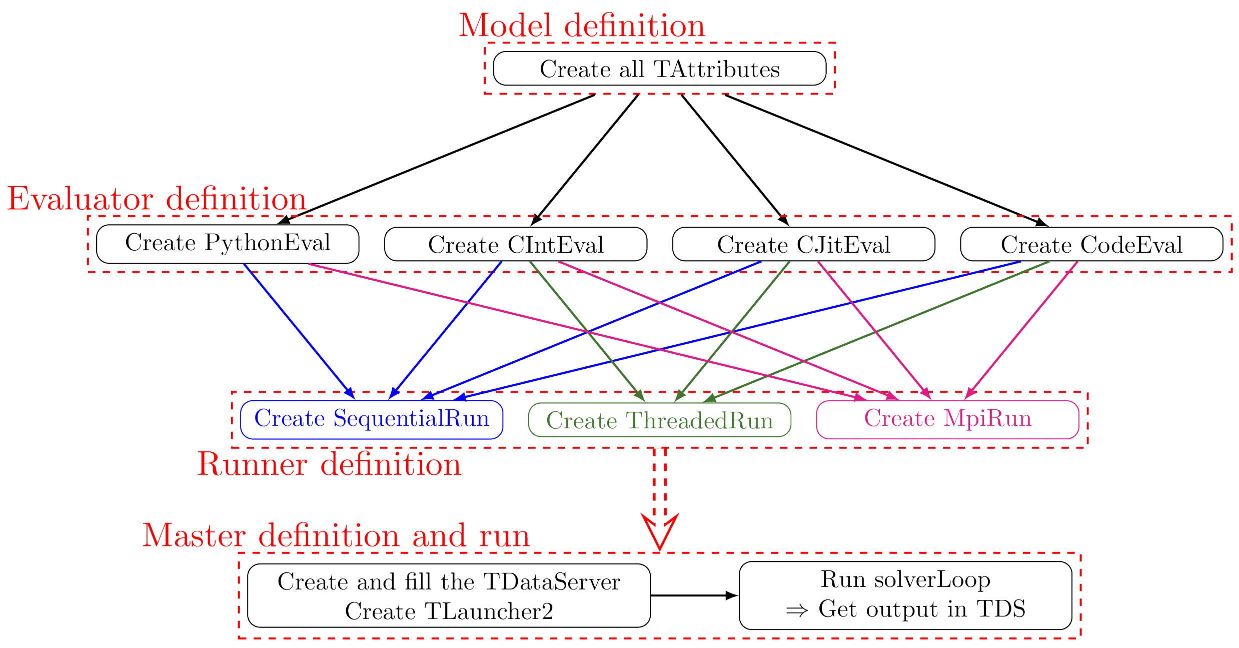


Figure VIII.1: Schematic description of the needed steps to define a relauncher procedure

The next three sections will successively pass through the different level of abstraction, starting from the bottom level to the top one.

VIII.3 TEval

The `TEval` abstract class defines the interface of user evaluation model. It provides an user class basis to define a model, and allows composition class to combine models together.

A standard evaluation is a function from n input parameters to m output parameters. n and m are known and fixed during the run, but they can now be string, vectors, or double. These evaluations may not return a value. Generally, it is due to an inconsistent input set. `TMaster` may support (or not) this lack.

A glimpse of all the assessor classes available can be found in Figure VIII.2 that displays the hierarchy of class starting from the inheritance of `TEval`.

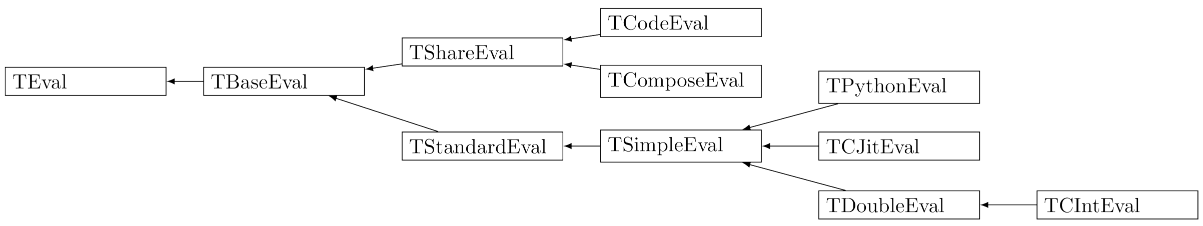


Figure VIII.2: Hierarchy of classes and structures for the evaluation part of the Relauncher module.

All the assessors will need to have attributes attached to them, both inputs and outputs. Input attributes might have a peculiar status, by being constant disregarding the provided stochastic law. On the other hand, output attributes might be considered temporary if, for instance, their value is of no interest for the final analysis but might be used by another assessor in composition. All this is discussed later on, in Section VIII.5 as these specification will be done directly on the master object.

VIII.3.1 TCIntEval and TCJitEval

VIII.3.1.1 function prototype

These classes reify the TLauncherFunction analytic function.

The ROOT on-the-fly compiler is now compatible with the C++ syntax but there are two ways to access an already written functions:

1. with a pointer to the function directly (this function being properly compiled);
2. through the ROOT-register of function, passed as a string only;

For the former, whose interface is the TCJitEval class, only usable in C++, there are no problem of thread-safety which is not the case for the latter, whose interface is the TCIntEval class which is not considered thread-safe. Given these consideration the latter shall not be used with a TThreadedRun, while the CJit version deals with the ROOT *Just in time* compiler, and is considered thread-safe. This is further discussed in Section VIII.4.2.

User supplied function needs to comply with a C prototype. Both version class constructors can use the TLauncherFuncti
prototype. Three extended prototypes are added for a more advanced usage.

```
// TLauncherFunction prototype
void standart_prototype(double *in, double *out);
// extension prototypes
int extended_prototype(double *in, double *out);
int extended_prototype(double *in, double *out, void *foo);
int uentry_prototype(std::vector<URANIE::DataServer::UEntry*> *in, std::vector<URANIE:: ↩
    DataServer::UEntry*> *out);
```

The first extension is used to deal with non-calculable evaluations. It uses the return value, which should be 1 in usual case and 0 if the output values cannot be computed. The second one can be used to pass extra information to the function by casting it as a pointer of void. The last extension prototype is used when evaluation deals with string or std::vector<double> input or output. Its use is tricky so we recommend to contact us if this is the only solution you have.

The class constructor first argument is the function pointer.

- TCJitEval can use standart and extended prototype and recognises it at compile time. This is the only runner able to deal with the complicated extended prototype.
- TCIntEval can use standart and extended prototype, but cannot recognise it (first argument is the function name as a string). It uses a second boolean argument to distinguish them. Default implicit value is kFALSE for standart prototype (kTRUE for first extended prototype, void * value for second extended prototype.

VIII.3.1.2 parameter declaration

The class definition gives information neither about the input or output number nor about parameter order. These information have to be added to link with TDataServer and TMaster definitions. We use the addInput and addOutput method with TAttribute objects as argument to do so. Inputs and outputs have to be added in correct order. Here is an example of how to precise inputs and outputs, from the script in Section VIII.2:

```
// problem variables
TAttribute x("x", -3.0, 3.0),
           y("y", -4., 6.),
           ros("rose");

// user evaluation function
TCIntEval eval("rosenbrock");
eval.addInput(&x); // Adding attribute in the correct order
eval.addInput(&y);
eval.addOutput(&ros);
```

For a more user friendly definition, you can use the `setInputs` and `setOutputs` methods. Instead of only requesting a pointer to the attribute under consideration, a first compulsory argument is an integer that equals the number of attributes to come, followed by as many pointers to attribute. Taking the example provided above, one can create a second assessor, using theses methods instead of the `addInput` and `addOutput` ones.

```
// user evaluation function
TCIntEval eval2("rosenbrock");
eval2.setInputs(2,&x,&y); // Adding attributes in the correct order, all at once
eval2.setOutputs(1,&ros);
```

VIII.3.2 `TPythonEval`

This class is used when you use Uranie with the python interpreter. So they are not detailed here. More detail can be find in the python version of this documentation, but its use is very similar to a `TCJitEval` class.

Its use with the C++ interpreter is not trivial (it needs to loads the python interpreter) and are not covered here. Using a `TCodeEval` instead is a simpler solution.

VIII.3.3 `TCodeEval`

This class takes up the `TCode` class features and make it thread safe.

Evaluation is done by an external executable, which reads one (or more) configuration file, where it finds its inputs, and writes the single result file, where to find the output variables. `TCodeEval` must create or adapt configuration files to introduce item values, run the executable, and analyse the result file to get its outputs. It needs to know file formats, and where to find values.

VIII.3.3.1 Local environment

In order to be used in parallel (MPI or thread), we have to take care of file access conflicts: many processes which modify the same file. To avoid such a thing, Uranie creates for each resource a personal directory named `URA` with then a set of numbers and letters that is more robust than the older version with just increasing numbers. Everything is done in it: input files are created there, executable is run from it and the output file are supposed to be found here as well. By default, these directories are created in the current folder. You can specify another root directory, using the `setWorkingDir` method. There are two other methods that can be called to change this:

- `setOldTmpDir()`: This will create folder named `URANIE0`, then `URANIE1` and so on, up to the number of process chosen (for sequential job, only `URANIE0` will be created).
- `keepAllFolders()`: This method is meant for debugging. It creates a specific working directory for EVERY computations (warning it might overflow your home directory).

Input files are often created from template files. These files, if they are not defined with a full path, are search in the current directory. You can specify another one using the `setReference` method.

The class constructor takes a string (`const char*`) as argument which is the command line used to launch the executable. `%D` jocker can be used, and will be replaced by the local directory.

VIII.3.3.2 Various file format

With the introduction of the vectors and strings from version 3.10.0, more complex interaction with files were introduced: how to differ two iterations of a single vector and how to differentiate a double from a string. This depends highly on the nature of the input/output file under consideration, whether it is just a text file used as database (in this case it depends mostly on the way you've written the code that generate/parse it) or whether it corresponds to a more strict kind of file, for instance a piece of code (c++/python/zsh). In the latter case, strings and vectors are not written in the same way. To take this into account, a rule has been defined (commonly to both input and output files, both in the Launcher and Relauncher module). There is a method for any kind of file to define properties of vector and string objects:

- `setVectorProperties(string beg, string delim, string end)`: the first element is the string beginning of the vector (usually "[" for python, "(" for zsh/sh, nothing...), the second one is the delimiter between iterators (usually "," for c++/python, blank for zsh/sh...) and the last one is the end of the string (usually the opposite character of the beginning one).
- `setStringProperties(string beg, string end)`: the first and second elements are respectively the beginning and ending character used for string (oftenly """).

Depending on the kind of chosen file, there is a default configuration chosen. This default is precised in the following two sections.

Warning
The chosen output format has to be consistent with the output parameters investigated, particularly when some are vectors which can be empty for some specific configurations. In this peculiar but still possible case, the abscence of results is indeed a result of its own and should not be taken as a failure (from an incomplete output file for instance, this specific aspect being further discussed later-on in Section VIII.5.2.1).
Most of the time this would be independent of Uranie as it would be specific to the code under investigation, and as such, it might be tricky to handle. Two use-case macro have been written to show this, so please take a look at empty vectors considered as results in Section XIV.8.11 or considered as an error in Section XIV.8.12 only because of the way the output Key-format output file is written. In a nutshell, in the former case caution has been taken to properly delimit and condensate the results so that even when the vector is empty there are sign of this, while on the other hand a simple dump is done for every instance of the vector meaning that with no content, no dumping is done leading to Uranie stating that there might be missing information in this output file (once again this is discussed in Section VIII.5.2.1).

VIII.3.3.3 Input file

Input file formats supported by `TCodeEval` objects, include:

- `TFlatScript`, Input file is created from scratch. Values are given in order separated by a blank separator. The default behaviour with respect to strings and vectors for this file, is to look like the DataServer format (from the Launcher module): strings have no specific beginning/ending characters, as for the vectors whose delimiter is chosen to be a comma.

- TLineScript, Input file is created from scratch. Each TAttribute values are written on a specific line. Changing attribute means changing line. It is the equivalent of the Column format (from the Launcher module).
- TKeyScript, Input file is created from an original file. Each TAttribute is associated to a keyword. Values are substituted using a *"keyword = value"* pattern.
- TFlagScript, Input file is created from a template file. Each TAttribute is associated to a keyword. Each keyword is substituted directly by the current value.

TXmlScript is not provided in this version

The addInput method is used to declare parameters for all these file types. For TFlatScript and TLineScript, it takes a single argument: a pointer to a TAttribute object, while in the two other cases, the same first argument is completed by a const char * for the key. The declaration order is only significant when no key is specified (so for the TFlatScript and TLineScript files).

```
// Input File Flat format case
TFlatScript finp1("input_rosenbrock_with_values_rows.dat");
finp1.addInput(&x); // Adding attributes in the correct order, one-by-one
finp1.addInput(&y);

// Or Input File Key format case
TKeyScript kinp1("input_rosenbrock_with_keys.dat");
kinp1.addInput(&x, "x"); // Adding attributes in the correct order, one-by-one
kinp1.addInput(&y, "y");
```

A more condensed version of this addInput method exists for all these input types: the setInputs method. Here as well, it takes an extra compulsory argument, an integer which equals the number of attributes to come. The rest of the arguments are either a single pointer to a TAttribute object for the TFlatScript and TLineScript objects, or a pair composed of the same pointer directly followed by the key (for the two other input file types). Starting back from our example written above, one could condensate this into these lines:

```
// Input File Flat format case
TFlatScript finp2("input_rosenbrock_with_values_rows.dat");
finp2.setInputs(2, &x, &y); // Adding attributes in the correct order, all at once

// Or Input File Key format case
TKeyScript kinp2("input_rosenbrock_with_keys.dat");
kinp2.setInputs(2, &x, "x", &y, "y"); // Adding attributes in the correct order, all at ↩
    once
```

Once done, the input files are provided to the TCodeEval object, using the addInputFile method, as shown below:

```
// Add to the TCodeEval
TCodeEval code("rosenbrock -r"); // put "rosenbrock -k" instead for key
code.addInputFile(&finp1); // put &kinp1 instead for key
```

VIII.3.3.4 Output file

Output file formats supported by TCodeEval include:

- TFlatResult, Output file is made up of an header characterised by # as first line character, and a line of floats separated by spaces. By default, it is constructed as the DataServer one (from Launcher module). One can consider using a flat output file written over several lines (so constructed as a TOutputFileRow) but one needs to be very careful about the fact that all attributes might not have the same number of entries (when dealing with vectors for instance). This is discussed in the third item of Section IV.3.1.2.3 and in Section XIV.4.32.1. To do this a specific method has to be called `isMultiLine(string separ)` which says to the class that the results are written over many lines, and every field is separated by the string `separ`.

- `TKeyResult`, Value can be found on line composed with the key, a separator, the value and eventually a ; character. A separator is composed with space, tab, = and : characters.
- `TLineResult`. All the values of a given `TAttribute` are written on a specific line. Changing attribute means changing line. It is the equivalent of the Column format (from the Launcher module).

`TXmlResult` is not provided in this version

In a similar way of `TInputFile`, one should use the `addOutput` to declare parameters, the argument being the pointer to the attribute under consideration for all these formats, pairing with the corresponding key when dealing with a `TKeyResult` object. This step can be gathered in a single operation, as for the input file, using the `setOutputs` method. Here is an example for the ongoing use-case.

```
// Input File Flat format case
TFlatResult fout("_output_rosenbrock_with_values_rows.dat");
fout.addOutput(&ros); // Or fout.setOutputs(1, &ros);

// Or Input File Key format case
TKeyResult kout("_output_rosenbrock_with_keys.dat");
kout.addOutput(&ros, "ros"); // Or kout.setOutputs(1, &ros);
```

Finally, use the `addOutputFile` method of `TCodeEval` to declare it:

```
// Add output file to the TCodeEval
code.addOutputFile(&fout); // put &kout instead for key
```

VIII.3.4 Evaluation functions composition

Composition offer the possibility to build an overall new kind of `TEval` from the succession of many others. It defines an ordered sequence of evaluation functions. One important thing to notice is that composition does not deal with distribution even if it is possible. It just applies sequentially all assessors and become really helpfull as the output of an assessor at the i-Th rank can be used as input for the next assessor, the (i+1)-Th one. This is one by creating a `TComposeEval` object as discussed briefly below.

VIII.3.4.1 `TComposeEval`

This composer can be seen as an overall new assessor that is, usually provided to the runner (or to a master directly).

The constructor have no argument. The only important method is the `addEval` one that allows users to add evaluation functions, keeping in mind that they should be called in the correct order, regarding what they expect (the only argument of the function being a pointer to the assessor to be stacked to create the chain). Examples of composition can be found in Section XIV.8.14.

As for the input/output attributes or a regular assessor, one can use the `setEvals` (mind the "s") that allows to put all the pointers as argument of the function, right after an **int** parameter that state the number of assessor procided to define the composition.

VIII.4 TRun

The TRun sub-classes deals with the use of computer resources. Three modes are available:

- TSequentialRun: evaluations are computed sequentially on a single computer core.
- TThreadedRun: evaluations are computed using the computer multi-core resources. It uses the *pthread* library with the shared memory paradigm. Using this runner prevents from using some assessor, as one should take care of memory conflict.
- TMpiRun: evaluations are computed using a network of computers (usually multi-core) It uses the *message passing interface (MPI)* library with a distributed memory paradigm.

If you run on a single node, you can use MPI or threads. MPI parallelisation is more expensive, but more generic (no thread safe problem).

Warning
Disregarding the chosen solution to distribute the computation as long as it is parallelised (meaning whether one is choosing thread or MPI) the number of allocated ressources (in the constructor or specify to the mpirun command) should always be strickly greater than 1. CPU number 1 will always be the "master" that is dealing with the distribution to its "slaves" and the gathering of all results.

The runner class hierarchy is smaller than the assessor one, as can be seen in Figure VIII.3. It starts with the TRun class, which is a pure virtual one in which few methods are given along with an integer to describe the number of CPUs.

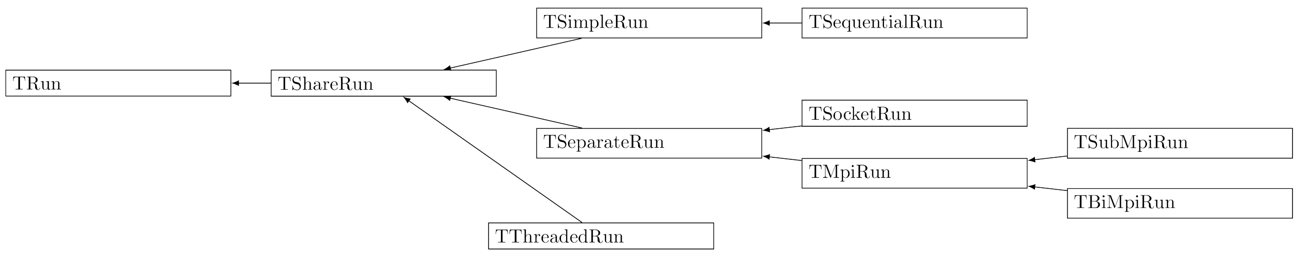


Figure VIII.3: Hierarchy of classes and structures for the runner part of the Relauncher module.

VIII.4.1 TSequentialRun

In this case, there is no distribution. If evaluations are fast, it remains the simplest way to run the evaluations. Here is the interpretation of the inherited methods:

- startSlave: exits immediately,
- onMaster: tests is true

- and `stopSlave`: cleans `TEval`.

`TSequentialRun` constructor only has one argument, a pointer to a `TEval` object.

```
// Creating the sequential runner
TSequentialRun srun(&code);
```

VIII.4.2 `TThreadedRun`

In this case, the program starts using a single resource (the main thread), then it launches evaluation on dedicated threads (children), uses them and stops them before ending.

Threads use a shared memory paradigm: all threads have access to the same address space. All objects that are used are defined by the main thread. Evaluation threads only use (or duplicate) them. It's only the main thread that follows the macro instructions, while its children only do the evaluation loop. Here is the interpretation of the inherited methods:

- `startSlave` starts some threads dedicated to evaluation (it is a unblocking operation), and then exits. These threads loops for evaluations.
- As we are on the master thread, `onMaster` is true.
- `stopSlave` puts fake items for evaluation. When the thread gets it, it stops their evaluation loop and exits. Main thread waits for all threads to be stopped.

`TThreadedRun` constructor has two arguments, a pointer to a `TEval` object and an integer. The second argument is the number of threads that the user wants to use.

```
// Creating the threaded runner
TThreadedRun trun(&code,4);
```

One important thing to take care is that the user evaluation function need to be **thread safe**. For example, with the old ROOT5 interpreter, the rosenbrock macro (see Section VIII.2) cannot be distributed with thread. This is because the user function is interpreted and the Root interpreter is not thread safe. You have to turn it in a compiled format to make it works with threads.

Thread safe problems come usually with variable affectation. If two (or more) threads modify the same memory address at the same time, the code expected behaviour is usually disturbed. It can be a global or static variable, an embedded object working variable, a file descriptor, etc. Thread unsafe bug is difficult to squash. It may be necessary to clone objects to avoid such problems.

Warning One might want to use `TDataServer` objects in code of `TCJitEval` instances that would be distributed with a `TThreadedRun` object. In this case, it is mandatory to call the method `EnableThreadSafety()` to remove all dataserver and tree from the internal ROOT register which would induce race-condition. This can be done as below:

```
ROOT::EnableThreadSafety();
```

An example of this (very specific) usage, is shown in Section XIV.8.3 for C++ mainly as it uses a CJit function which cannot be used in python.

VIII.4.3 TMpiRun

In this case, many processes are started on different nodes. MPI uses the distributed memory paradigm: each process have is own address space. All processes run the same macro and define their own objects. If you create a big object in the evaluation/master code section, all processes allocate it (this is why, generally, the main dataserver object is created in the onMaster part to prevent from creating as many dataserver as there are slaves).

- the constructor calls MPI_Init for the initial process synchronisation. This step is automatical, as long as one is running through the on-the-fly C++ compilator thanks to the root command or in python. In the peculiar case of standalone compilation please refer to the provided exemple and the discussion on how to handle this in Section XIV.8.8.2.
- startSlave either exits immediately for the master process (id=0) or starts evaluation loop for other ones.
- depending if we are on the master process or not, onMaster is true or false.
- stopSlave puts fake items for evaluation and then exits. Evaluation processes get it, stop their loop, exit from startslave, and usually jump the master bloc instructions. Unlike threads, the master process is not waiting for evaluation processes.
- the destructor calls MPI_Finalize for the final process synchronisation. .

TMpiRun constructor has one argument, a pointer to a TEval object.

```
// Creating the threaded runner
TMpiRun mrun(&code);
```

To run a macro in a MPI context, you have to use the mpirun command. Here is a simple way to run our example:

```
 mpirun -n 8 root -l -q -b RosenbrockMacro.C
```

Here, we launch root on 8 cores (-n 8); -q option (quit) is needed to exit the ROOT interpreter at the script end; -b option (batch) is needed when running on many nodes, preventing opening display. The **mpirun** command has other options not mentioned here.

In general, one runs a MPI job on a cluster with a batch scheduler. The previous command is put in a shell script with batch scheduler parameters. The ROOT macro does not use viewer, but saves results in a file. They will be analysed in a post interactive session using all the ROOT facilities.

If one wants to run in a compiled way, this cannot be done just by adding a "+" to the command line. Effectively, if all processes try to compile using the same output file, conflicts occur. One way to do is to run a first ROOT session without mpirun to compile your macro. Then, if you run a second mpi root session with the single "+", processes will use the pre-compiled macro. You can compile your macro with the command:

```
gROOT->LoadMacro("Rosenbrock.C++");
```

LoadMacro compiles it but does not execute it. Another possibility to run a code in a compile way is to consider the standalone compilation which consists in considering Uranie as a set of libraries, as already discussed in Section I.2.3.

Warning
The TMpiRun implementation requires also at least 2 cores (one being the master and the other one the core on which assessors are run). If only one core is provided, the loop will run infinitely.

VIII.4.3.1 **TBiMpiRun and TSubMpiRun**

In some case, users want to use multi level of parallelism. Two examples are given in the use cases section : first one is an optimization where each evaluation realizes an experiment design and launchs many evaluations and returns a overview of values (max, min, mean) ; second one uses an MPI function (`TCJitEval`) for evaluation.

For a two level MPI, two classes are provided : `TBiMpiRun` and `TSubMpiRun`. `TBiMpirun` is the high level class and splits MPI resources in different parts : one ressource for the `TMaster` and n resources for each `TEval`. `TSubMpiRun` gives acces to the n ressources reserved for evaluation. For example with 16 resources, 1 resource is reserved for the master and the rest can be splited in 3 parts of 5 resources each for evaluation. `TBiMpiRun` got an extra parameter, an `int` defining the number of each evaluation resource. This number must be compatible with available resources (with 16 resources, it could be only 3 or 5).

VIII.5 TMaster

The object inheriting from the `TMaster` class has a supervisor role: it defines items to be evaluated, and treats these evaluations. Usually, it is created with a `TDataServer`, and a `TRun` (`TEval` implicitly) that is used for evaluations. The `TMaster` retrieves information both:

- from `TDataServer`, it extracts the declaration of the input variables.
- from `TEval`, it extracts extra parameters that will be computed for each item.

We could distinguish two cases depending of the `TDataServer` data:

- if it contains a header and data, the `TMaster` will complete items data by adding information (columns). This happens most of the time when considering the pure launching aspect, meaning when using the `TLauncher2` object, briefly defined below.
- if it contains only a header, it will be used as items definition parameters. The `TMaster` will fill it with its own items (lines and columns). This case happens both for pure the launching aspect, given that a design-of-experiments will be constructed at some point, but also for optimisation issues.

Disregarding the case, `TMaster` interface defines an abstract method, `solverLoop`, that runs the evaluation loop and completes or fills the `TDataServer` data. This method is the one to properly start the analysis.

VIII.5.1 Dealing with attributes

Disregarding the kind of master considered (the `TLauncher2`, discussed briefly below or those introduced in the optimisation part, in Section IX.2 for instance), there are common methods that are discussed below in order to simplify or precise some important behaviour of your current analysis.

In simple problem, the `TDataServer` object and the ones deriving from `TEval` have input parameters that fit together. In this case, you can use the `addAllInputs` method from the `TEval` class to define the `TDataServer` needed header (the only argument being a pointer to the considered dataserver object). For more complex one, there might be a mismatch.

In other problem, one can distinguish between three kinds of situation:

- The `TEval`-based object is using few input parameters, one (or more) of which is requested to be set a constant value for all the upcoming estimation. To do this properly, the parameter under consideration should be added to the assessor but not to the dataserver. The method `addConstantValue` from the chosen `TMaster` inheriting object is called with argument the pointer to the attribute and its value as second parameter. A third optionnal parameter (set by default at false) specifies whether this value should be stored in the dataserver tuple at the end. This option has been recently added as it might be better for bookkeeping to know what the underlying hypothesis was when the computations were performed. An example of how to use this is provided in Section XIV.8. This method has also been implemented to be able to cope with vectors and strings but this implies to handle complicated concept that are used internally, so we strongly recommand to contact us if you have no other choice.

- The `TEval`-based object (or more oftently the composition of evaluators) is using many internal parameters (attribute that are created from one assessor and used by any other later on in the chain). If these parameters are uninteresting for the final purpose of the analysis, they can be removed, simply by calling the `addTemporary` method from the chosen `TMaster` inheriting object. The only argument is a pointer to the attribute under consideration. Once more, an example of how to use this is provided in Section XIV.8.

- Finally, in some pure launching problem (Sobol for instance), some `TDataServer`-objects have extra parameters unused by the `TEval`. In this case, it means that the initial `TDataServer` is not empty and that the `TMaster`-object is able to keep this parameters values, and completes the data by adding columns.

VIII.5.2 TLauncher2

`TLauncher2` is a new version of `TLauncher` and `TLauncherFunction` using the relauncher classes. It's a basic implementation, and it has no extra method. The constructor takes the `TMaster` usual arguments: a `TDataServer` and a `TEval`.

VIII.5.2.1 Dealing with failure

From time to time, there might be problem when running a code. The source of these problems might differ from one instance to another. In the Relauncher module one can consider different case:

- the command is returning a failure code. A convention for a command is to return 0 if all went well which means that what should be done has been done while any other returned value can be considered as a way to inform that a problem has been met. From the Uranie point of view, there is no way to know what kind of error is meant as it is command dependent.

- the `system` function is returning other values. The command provided by the user is passed to the `system` C++ function and on top of the non-zero returning value (discussed above), there are few specific cases. Among these, one can find the case where the command is not known (the user made a mistake in the `TCodeEval` definition) or the command is suffering from an internal problem preventing from even reaching the exit (a segmentation fault for instance). With most Linux platforms, the former returned code is 127 while the latter is 139. This interpretation is just given for illustration purpose but unfornutatly no generalisation can be done.

- no output file is created. In this case, nothing can be done, all output variables will be missing and this might arise even if the code is not returning any specific failing code (see the discussion above).

- One or more output variables are missing in the output file. This is special, as it can arise if the command has stopped during the writting process, but this also can be coming from an empty vector whose output formatting has not properly being taken care of. This is discussed in Section VIII.3.3.2 and illustrated in a use-case macro Section XIV.8.12 because of the way the output Key-format output file is written

All these problems will be considered as a failure from the command. As such, the input configuration will be discarded and not stored in the final dataserver object. The following paragraph discussed the way to get back all failed configurations.

Unlike in Section VIII.5.1 where all the methods are inherited from the `TMaster` class, the method `setSaveError(TData *tdserror)` has been implemented in order to help handling the command failures discussed previously. Its only argument is a pointer to a dataserver object in which all failing configurations will be stored. Example of its usage is shown in two use-case macros with the classical flowrate case in Section VIII.3.3.2 and illustrated in a use-case macro Section XIV.8.9 and when an empty vector is interpreted as a missing information because of the wrong output file formatting, see Section XIV.8.12.

Chapter IX

The Reoptimizer module

IX.1 Introduction

The reoptimizer module provides optimisation features, using the relauncher architecture. It can indeed be considered as a specialisation of this module for the closed-loop case (optimisation steps usually depends on the previous steps). This document will present neither an introduction to the optimisation problem characteristics (one simple version can be found in [30]), nor a description of the different algorithms. It will more discuss the possible combination of runners, solvers and masters with (if possible) their pros and cons. For a glimpse at the way the implementation of a script can be done, one can look at the use-cases, provided in Section XIV.9.

Two kinds of solver are proposed: local search ones, starting from an initial guess point and global search ones, starting from a random population of potential solutions. They differ on many points:

- Problem types: only global solvers offer multi-objective optimisation.
- Constraints: Global solvers deal with inequality but not equality constraints, while local ones deal with both of them.
- Inputs nature: all solvers deal with continuous problems, and some global solvers can deal with combinatory problems, with extra works.
- Convergence speed and robustness: local solvers need less evaluations to converge but can be trapped in a local minimum, while global ones are more robust.
- Unavailable items: only global solvers can deal with it.
- Parallelism: local solvers are intrinsically sequential, and the only way to parallelised them is to run many optimisations starting from different points. Global solvers are intrinsically parallel.

IX.1.1 local optimizer

A local solver proposes mono-objective optimisation with or without constraints. It uses the NLopt library [35] and different solvers are proposed. These solvers could be distinguished with:

- Needed information: some solvers need gradient information while others use just the objective value.
- Constraints handling: some solvers handle constraints naturally while the others use the augmented Lagrangian method.

IX.1.2 global optimizer

A global solver proposes mono and multiobjective optimisation, potentially with constraints. It uses the Vizir library [29] and offers different solvers.

IX.1.3 Number of objectives

The number of objectives to be minimised plays a crucial part in the technique to be used, along with the time needed for a given code to converge and provide results. The main idea behind this small discussion is to identify the needs when starting an analysis.

- One objective or more than one but they are not antagonistic: a single objective optimisation can be done. When dealing with several criteria the idea is then to combine them into an homemade criteria, based on whatever recipe one wants to apply (weighted/unweighted criteria, L1 or L2 sum, ...). From there, the use of `TNlopt` algorithm is recommended.
- More than one objective that cannot be combined. In this case, the approach recommended would be to use Vizir which contains multi and many objectives algorithms. The difference between multi and many is tidious and is oftenly set between 3 and 4. A discussion between these algorithm is provided in [30].

IX.2 Problem definition

An optimisation problem is a kind of parametric studies, so, it could make the best of the Relauncher architecture. Having this in mind, having a look at Chapter VIII is crucial in order to get good understanding of the following, already-introduced, concepts.

In this module, one will introduce new *masters*, meaning classes that inherit from `TMaster` which will handle the distribution of evaluations, with extra specificities linked to their optimisation purpose. But before using them, the *runners* (inheriting from `TRun`) and *assessors* (inheriting from `TEval`) have to be defined as well, respecting the pattern separation between study and evaluation sides. Only the study side is concerned by optimisation objects (all these notions are defined in Chapter VIII). The standard steps to solve an optimisation problem are:

- to declare the optimisation input parameters.
- to choose a solver (eventually configure it).
- to create a master (eventually configure it).
- to declare the objective and constraints of the problem.
- to run the optimisation.
- to analyse the results.

The Rosenbrock example script provided in Section VIII.2 gives a simple example of these steps.

There are few `TMaster` sub-classes depending of the local or global algorithm that is chosen (the underlying library). The constructor has three arguments: the two `TMaster` usual arguments (a `TDataServer` and a `TRun`), and the solver. A common method to all masters is `setTolerance` with a double as only argument. It defines a threshold to stop the search. However, its interpretation is solver dependant.

As we saw in the TMaster section, item definition parameters are defined in the `TDataServer` used as first constructor argument. These attributes generally need to be defined with a domain, whose boundaries are used for the optimisation.

The master and solver declaration will be covered in next section. Running the optimisation is done by the `solverLoop` method, and results will be found in the `TDataServer`.

Tip
Before Uranie version 4.2, only the final results were kept in the dataserver and no option was allowing the user to keep track of all performed estimation (either to see how the algorithm is driving the parameters evolution, or just for bookkeeping). From version 4.2, it is possible to create an empty `TDataServer` and to provide it to the chosen `Master` so that every computation will be stored in this specific object. For a single objective optimisation this should look like this:

```
// ... Problem definition
runner.startSlave(); // Usual Relauncher construction
if(runner.onMaster())
{
// Create the main TDS
TDataServer tds("nloptDemo", "Param de l'opt nlopt pour la barre");
tds.addAttribute(&x);
tds.addAttribute(&y);

// Defining the optimisation condition
TNloptCobyla solv; // algorithm

// Create the single-objective constrained optimizer master
TNlopt opt(&tds, &runner, &solv);
// ... + objective, constraint...

// Create the dataserver in which all computation will be stored
TDataServer trc("allevents","dataserver containing all events");
opt.setTrace(&trc); // pass the dataserver to the master

opt.solverLoop(); //perform the optimisation
}
```

IX.2.1 Objectives and Constraints

An optimisation problem is defined by an objective (may be more for multi-objective problems) and eventually some constraints (objectives can as well be called criteria in various literature). An item evaluation may return many values. Some of them may be used as objectives or constraints, while the others are left unused by the solver. The master methods `addObjective`, `addConstraint` and `addEquality` may be used to declare the corresponding values. The last method is only available in local solver. All these methods have a first argument, the output variable (a pointer to its corresponding `TAttribute` object), and a second optional argument (a pointer to a modifier object).

Modifiers are used to overwrite the default solver behaviour: objectives have to be minimised, constraints are satisfied when their values are lesser than zero, and equality when their value is zero. Once this is settled and when the returned value does not fit with these defaults, a modifier have to be used. Existing modifier classes are:

- `TMaximizeFit` objective modifier: value has to be maximised.
- `TTargetFit` objective modifier: value has to be closed to a target value.

- `TLesserFit` constraint modifier: value has to be lesser than a threshold value.
- `TGreaterFit` constraint modifier: value has to be greater than a threshold value.
- `TInsideFit` constraint modifier: value has to be inside a domain.

The chosen threshold value(s) are passed in the constructor.

Warning
In the current implementation, it is not allowed to use an input variable as an objective or a constraint.

IX.2.2 Sizing of a hollow bar example problem

In order to give a more detailed example of the usage of both local and global solvers, the hollow bar problem is introduced. It consists in finding the lengths of the internal and external sides of a hollow bar with a square section, minimising its weight (*i.e.* its section) and its deformation by an external force applied at its centre. The two lengths are normalised so that they evolved in a 0 to 1 range and the pipe can be sketched as done in Figure IX.1.

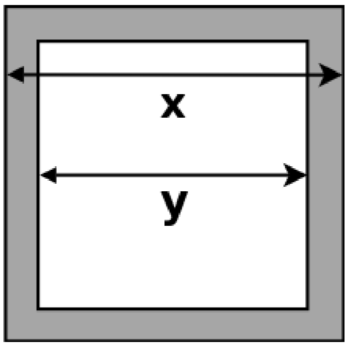


Figure IX.1: Hollow Bar

The problem has then three variables:

- $f_1(x,y) = x - y$ which is called the thickness.
- $f_2(x,y) = x^2 - y^2$ which is called the section.
- $f_3(x,y) = (x^4 - y^4)^{-1}$ which is called the distortion

and three natural constraints:

- $0 < x < 1$.
- $0 < y < 1$.
- $x > y$

In the following sections the idea will be to study (if possible) the minimisation of the section of the bar and the distortion, keeping a minimum thickness of about 0.4. This threshold is chosen so that the bar can sustain its own weight. The examples will use an external code to compute the three previously-introduced variables once both the internal and external lengths are provided. The code is written in python as following:

```python
#!/usr/bin/env python
"""
Simple file to mimick the barAllCost function to emulate it as a code
"""

x = .8
y = .3


def barre(out_l, in_l):
    """Compute the constraint and objectives for the hollow bar
    Arguments:
    out_l -- outter length of the hollow bar
    in_l -- inner length of the hollow bar
    """
    epais = out_l - in_l
    surf = out_l*out_l - in_l*in_l
    defor = 1 / (1.e-66 + out_l*out_l*out_l*out_l - in_l*in_l*in_l*in_l)
    return [epais, surf, defor]


def echo(out_l, in_l):
    """Print the results of the hollow bar computation
    Arguments:
    out_l -- outter length of the hollow bar
    in_l -- inner length of the hollow bar
    """
    print("#COLUMN_NAMES: c1|o1|o2")
    print("")
    evt = ""
    for i in barre(out_l, in_l):
        # print i,
        evt += "%.25g " % i
    print(evt)


echo(x, y)
```

In both examples, the same tip is used: the *bar.py* file is used to perform the computation (as the code is defined by `python bar.py > output.dat`) but it is also defined as the input file for the `TCode`. Thanks to this, the file is copied to every temporary working directory (so no need to change the `$PYTHONPATH` environment variable) and no extra file is needed to define the inputs. The output is directly stored as an ASCII file compatible with the usual Salome table format so that it can easily be read and convert as a `TDataServer`

IX.3 Local solver

Local solvers proposed by Uranie come from the NLopt library (Non Linear OPTimization). This library is developed by Steven G. Johnson, integrating many original algorithms in an uniform interface. The NLopt website provides many useful information, that enhance the succinct ones provided here. The example used as support for this part is given in Section XIV.9.1 with a specification with respect to the way it was described in Section IX.2.2: since there is no multicriteria solver implemented from NLopt, the criteria on the distortion is downgraded to a constraint using a randomly defined threshold (set to 14).

IX.3.1 TNlopt

The `TMaster` subclass for local optimizer is called `TNlopt`.

Local optimization starts from an initial guess point. This point has to be defined using the `setStartingPoint` method with an integer to precise the dimension, as first argument and a `double *` as second argument. This has changed in order to be complient with python but also to be able to check that the provided number of double is matching the dimension of our problem under consideration. You can call the `setStartingPoint` method many times. In C++, it could look like this:

```
TNlopt opt(&tds, &runner, &solv);
...
vector<double> p{0.2,0.3};
opt.setStartingPoint(p.size(),&p[0]);
```

In this case, each optimization, starting from a corresponding starting point, may be done in parallel using an appropriate `TRun`. If the results of the optimisation is not consistent when changing the starting points, this might be a sign for a problem with local minimum. In this case a safer (but slower) solution might be to consider using global solvers.

To modify the default configuration, the `setMaximumEval` method, with an `int` as only argument, may be used. The default value is 10 000. For multi starting point, it is interpreted as the accumulation of all evaluations.

IX.3.2 Solvers

Uranie proposes different Nlopt solvers. For direct solvers, there is:

- `TNloptCobyla`: *Constrained Optimization BY Linear Approximation* from M.J.D.Powell works. The only direct algorithm that supports constraints naturally.
- `TNloptBobyqa`: *Bounded Optimization BY Quadratic Approximation* from M.J.D.Powell works. It is usually quicker than the previous one, but might give nonphysical results if the problem should not be assumed to be quadratic.
- `TNloptPraxis`: It uses the *PRincipal AXIS method* of Richard Brend. This algorithm has a stochastic part.
- `TNloptNelderMead`: The well known *Nelder-Mead Simplex* algorithm.
- `TNloptSubplexe`: a simplex variant, the Tom Rowan's *subplex* algorithm.

and for gradient-based solvers:

- `TNloptMMA`: *Method of Moving Asymptotes* from Krister Svanberg works. It deals with nonlinear inequality constraints naturally.
- `TNloptSLSQP`: *Sequential Least-Squares Quadratic Programming* from Dieter Kraft. It deals with both inequality and equality constraints naturally.
- `TNloptLBFGS`: an implementation of the *Limited memory Broyden-Fletcher-Goldfarb-Shanno* algorithm written by Ladislaw lukan.
- `TNloptNewtown`: A preconditioned inexact truncated *Newtown* algorithm written by Ladislaw Lukan.
- `TNloptVariableMetric`: An implementation of *shifted limited-memory variable-metric* by Ladislaw lukan.

You may notice that actually none of Nlopt global solvers is provided.

Tip

Remarks for the gradient case: oftenly one does not have access to the real gradient. In this case a finite difference method is used by doing $2n_X + 1$ computation around every point, which implies:

- a code that will provide results with a sufficient accuracy so that the gradient is not always assumed to be null (which can happen when a code returns a value with a very low number of digits). Even with a non-null estimation, the gradient accuracy needs to be sufficient since gradient-based solver usually estimates the Hessian matrix.
- a possible parallelisation of the computation.

IX.4 Global solver

$\mathbb{R}^p$ spaces have no ordering relations when p is greater than 1. As a consequence, finding the minimum of a function the output of which is defined in such a space has no meaning. Multicriteria optimisation solves this problem by finding the inputs corresponding to the *Pareto frontier* in the costs space.

The evolutionary algorithm library **Vizir**, allows to perform multicriteria optimisation with a global approach, and is available in Uranie. It is first introduced in a broad picture (even though more details can be found in [30]) before considering the different solvers available and their possible configuration. We have also illustrate its use in a simple example that can be found in Section XIV.9.3.

IX.4.1 A step-by-step description of Vizir

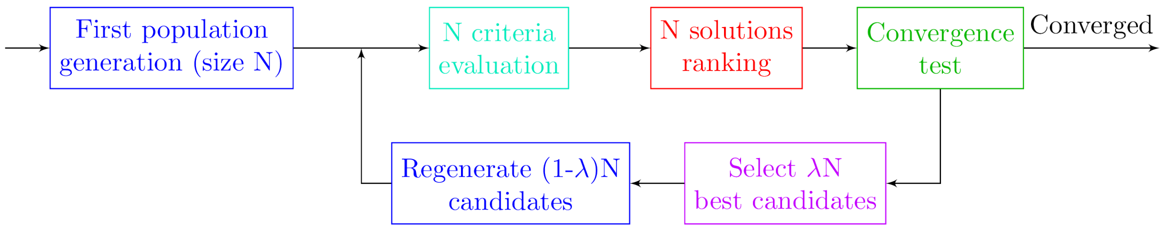

First population generation (size N)
N criteria evaluation
N solutions ranking
Convergence test
Converged
Select λN best candidates
Regenerate (1-λ)N candidates

Figure IX.2: Schematic description of the requested steps of an optimisation procedure once this one is performed with Vizir

The global organisation of an analysis performed within Uranie with the Vizir package is as followed: the user request a certain number N_{Cand} of elements to describe correctly the Pareto set and front.

1. The first step is to create randomly, only using the research space definition, a population of the requested size. An evaluation is performed for all candidates, meaning that the criteria and constrains will be tested and the results will be stored in a vector for all candidates. All the calculation concerning a candidate will count as one evaluation (this notion will be important later on when considering the number of evaluation N_{Eval}). This step (the turquoise blue box in Figure IX.2) is followed by the ranking of all the candidates (red step in Figure IX.2).
2. As already discussed, ranking single criterion results is simple because there is an obvious relation order, but this is not the case when dealing with multi-criteria. The chosen solution in the Uranie implementation is to affect a rank to a candidate under study, corresponding to the number of other candidates that dominate it. The best candidates have then a rank 0 (they are not-dominated), following by rank 1, rank 2... With the ranking step completed the next step is to test whether the algorithm has converged or not.

3. The test of convergence can reach three possible states:
   - all the tested candidates are not-dominated. This means the algorithm have converged and the loop will stop as the objective of having a description of both Pareto set and front is achieved.
   - not all candidates are not-dominated but the maximum number of evaluation has been reached. In this case, the algorithm stopped as well but this time without having converged. Restart the algorithm with different configuration might be a possibility.
   - not all candidates are not-dominated and the maximum number of evaluation is not reached.
4. In the latter case, a certain fraction of the candidates will be selected and used as a new starting point to recreate a new population. A fraction λ is indeed kept (this fraction value being set by changing the survival rate in the genetic case, whose default value is 40%) in order to produce a new generation that will hopefully converge better than the current one. The evolutionary algorithm uses the selected fraction (λN_{Cand}) to complete the total population (meaning re-generating $1-\lambda N_{\text{Cand}}$). This procedure is explained more deeply, in the case of a genetic algorithm, in [30].

IX.4.2 TVizir2 and TVizirIsland

The `TVizir2` and `TVizirIsland` classes are the `TMaster` subclass for global optimizer, the former being the more commonly used disregarding the chosen solver and runner. The latter is dedicated to the *island* principle: instead of growing a large population, N_{Is} populations are defined (this parameter can be changed by the dedicated `setIsland(int)` method) and are growing independently, exchanging inhabitants from time to time.

The `setSize` may be used to configure this. The first argument defines the population size (default to 250). The second optional argument defines the maximum number of evaluations that the solver may use (default to 100 000). The third optional argument defines the number of items born to create a new generation (default to 50).

IX.4.3 Solvers

The Vizir solvers, that are interfaced with Uranie, are:

- `TVizirGenetic`, a *genetic algorithm* using a diploid representation.
- `TVizirSwarm`, a *particle swarm* algorithm.
- `TVizirSimplex`, a *Nelder Mead Simplex* variant, adapted to work with a population, and to deal with multiobjective problem.

IX.4.3.1 TVizirGenetic

The `TVizirGenetic` (whose principle and vocabulary is define in [30]) can be configured using the following methods:

- `setMutationRate`, defines the rate of population items that are muted during creation. Default value is 0.01 (1%).
- `setHomozygoteRate`, defines the rate of population considered as homozygote. Default value is 0.5 (50%).
- `setSurvivalRate`, obsolete, defines the rate of population items that survive at each generation. use setSize instead. Default value is 0.4 (40%).

IX.4.3.2 TVizirSwarm

The `TVizirSwarm` behaviour can be configured using the following methods:

- `setLocalSize`, defines the particle memory size: its size and a generation step size.

The particle number is defines using the third parameter of `setSize` method (generation item number). It is half of it. No specific method is actually provided.

IX.4.3.3 TVizirSimplex

The `TVizirSimplex` have not any specific configuration method.

IX.4.3.4 Many objectives methods

The global solvers discussed here have dedicated methods to call many-objective algorithms. An example of implementation can be shown in Section XIV.9.4.

- `setMogaDiversity(int val=0)`
- `setCrowdDiversity(int vois=0)`
- `setPairDiversity(int vois=0)`
- `setIbeaDiversity(double k=0)`
- `setKneeDiversity(int vois=0, double taux=0.0);`
- `setMoeadDiversity(int cut1, int cut2=0, int vois=0)`
- `setStoppingCriteria(int stop=0)`

Chapter X

The Metamodel Optimization module

X.1 Introduction

For an optimization problem, when the evaluation function costs in term of resource, chaining the construction of a *surrogate model* with the optimization process could be a good solution. Uranie provides modules that can be used directly to do so. The Metamodel Optimization module gives another way to do similar things by coupling the two process: the *surrogate model* construction and the optimization research.

So far, the module provides a parallelized version of the Efficient Global Optimization (EGO) algorithm for mono objective problems.

X.2 Efficient Global Optimization

X.2.1 Introduction

EGO[17] makes a global search. As a genetic algorithm, it needs an adequate numbers of initial evaluated items to initiate its search: in our case, to be able to construct a sufficiently pertinent model. After this first phase, it builds a surrogate model, and then loops on updating the model with new evaluations, and on searching a new promising solution to evaluate using this model.

For its surrogate model, EGO uses kriging models which provide, for estimated points, a prediction value and an associated variance. EGO defines an objective, the expected improvement, which takes both of them into account and provides a trade-off between a good estimation value and a large uncertainty.

Because of its efficiency in term of evaluation number, this kind of algorithm is well suited when evaluations are expensive to compute. EGO algorithm is expensive: both construction of the *surrogate model* and search of the next attractive point are complex optimization problems, and are done many times, slowing down the problem resolution.

Extensions to constraints and/or to multi objective should come later in Uranie.

X.2.1.1 Parallelism

Uranie's EGO provides an asynchronous parallelism. With *n* resources, synchronous parallelism generates *n* items, evaluates them and waits for all results and iterates. In asynchronous parallelism, when a result comes, a new item is generated which takes into account the *n-1* evaluations in progress.

Usually a new point generation is not expensive and the resource that finishes its evaluation usually waits for it. It is not the case for EGO. To avoid wasting computation time, different approaches are implemented:

- the next point is generated before getting the evaluation result back (and so with n ongoing evaluations instead of n-1)
- after a point generation, if more than one evaluation are available, we get all results back, and if the solver afford to do so, we generate few points to be evaluated.
- otherwise, if no evaluation is available, the search of next attractive points are extended to try to improve them.

X.2.2 Problem definition

This module extends the Reoptimizer module and is kept in a separated module because of its dependency with the Modeler module. Its use follows the same structure and is also based on the Relauncher architecture. Having that in mind, it is suggested to have a look at Chapter VIII and at Chapter IX for a better understanding. As it uses a kriging model, it is also suggested to look at Section V.6.

The principal difference is in the solver definition: we have to define the kriging model and its construction parameters; and to use an adapted optimization solver.

X.2.2.1 TEGO

The `TMaster` subclass for EGO is `TEGO`. Its constructor has two standard arguments, a `TDataServer` and a `TRun` pointer. Three kinds of `TDataServer` can be passed to the class :

- An empty tds with the input `TAttribute` declared: initial points are random;
- A tds filled with a sampler where input `TAttribute` are declared: initial points are defined by user but they need to be evaluated.
- A tds filled with a launcher or relauncher where both input and output `TAttribute` are declared: initial filling phase can be skipped;

The `TEGO` objects have a method named `setSize` with two integer arguments: the first one, used in the case of an empty tds, gives the number of random points needed for the construction of the first *surrogate model*; the second one gives the maximum evaluation number of the expensive code.

Two different solvers can be defined, one for the *surrogate model* construction (using `setModeler` method) and one for the next point search (using `setSolver` method). If they are not defined a default solver is used.

Optimization loop ends when either:

- max number evaluation is reached;
- expected improvement objective is lower than a threshold;
- a Cholesky decomposition problem occurs in the *surrogate model* construction.

In these version, results are not filtered: all evaluated points are saved in the `TDataServer`

X.2.2.2 TEgoKBModeler

There is only one modeler currently available `TEgoKBModeler`

To deal with solutions that are currently under evaluation by the cpu resources (asynchronous parallelism), this modeler uses the *kriging believe* principle (it trusts in the model prediction). Two models are created: a first one, built with all the evaluated solutions, is used to estimate solutions under evaluation; a second model, built with both evaluated solutions and estimated ongoing solutions, is used by the solver to find the next solution to evaluate. Predictions is not significantly affected in the second model but the variances are, especially around ongoing solutions. The EI objective takes it in account, naturally driving next solutions away from them.

As it is an optimization, advancing in its search, EGO will generate solution near existing ones. It is a difficulty for model construction that can leads to Cholesky decomposition errors. To get around this problem, users can use the kriging regulation.

it uses

X.2.2.2.1 TEgoKBModeler

The `TEgoKBModeler` has a constructor without argument and 2 user methods:

- `setModel` defines the model that will be used in optimization. It has 3 parameters: a `const char*` defining the model to use (`"matern7/2"` for example); another `const char *` defining the trend (`"const"` for example); a `double` defining a regularization (`1.e-8`, use 0.0 for no regularization).
- `setSolver` define how the model will be constructed. It has 4 parameters: a `const char *` defining the objective to minimize (`"ML"` for maximum likelihood); a `const char *` defining the NLopt solver to use (`"Bobyqa"` for example); an `int` defining the size of the preliminary screening; an `int` defining the maximum evaluation number for optimization.

Take a look at Section V.6 for details on possible parameter values.

X.2.2.3 TEgoSolver

There are 4 available solvers combining two distinct features:

- dynamic or static optimization: in static, the search restart with a new random population; in dynamic the search restart from the previous population which should be rich enough to find a point that was put aside. In dynamic, the first search is longer than the following one.
- genetic or HJMA algorithm;

This leads to the following classes: `TEgoDynSolver`, `TEgoStdSolver`, `TEgoHjDynSolver` and `TEgoHjStdSolve`

some methods are provided:

- `significantEI` with a `double` parameter is used to define the EI threshold to stop the search loop
- `setManyNewItem` with an `int` is used to define the maximum number of new items used to feed the empty resources (unused with TEgoStdSolver).

- for the class using HJMA, `setSize` with two `int` parameters defines the number of global search performed by the optimization in the preliminary search and in the following ones.
- for the class using Vizir algorithm, `setSolver` with a pointer on a `TVizirSolver` defines the solver to use.
- for the `TEgoDynSolver`, the first and longer search uses the maximum evaluation number defined in the solver. The following search are shorter and is defined using `setStepSize` and its `int` argument.

Chapter XI

The Calibration module

XI.1 Introduction

This section presents different calibration methods that are provided to help get a correct estimation of the parameters of a model with respect to data (either from experiment or from simulation). The methods implemented in Uranie are going from the point estimation to more advanced Bayesian techniques and they mainly differ from the hypothesis that can be used. They're all gathered in the in the **libCalibration** module. The namespace of this library is **URANIE::Calibration**. Each and every technique discussed later-on is theoretically introduced in [30] along with a general discussion on calibration and particularly on its statistical interpretation.

The data provided as reference will be compared to model predictions, the model being a mathematical function $\mathbf{f}_\theta : \mathbb{R}^{n_X} \rightarrow \mathbb{R}^{n_Y}$. From now on and unless otherwise specified (for distance definition for instance, see Section XI.1.1) the dimension of the output is set to 1 ($n_Y = 1$) which means that the reference observations and the predictions of the model are scalars (the observation will then be written y and the prediction of the model $f_\theta(\mathbf{x})$).

On top of the input vector which is problem-dependent, the model depends also on a parameter vector $\theta \in \Theta \subset \mathbb{R}^p$ which is constant but unknown. The model is deterministic, meaning that $f_\theta(\mathbf{x})$ is constant once both $\mathbf{x}$ and θ are fixed. In the rest of this documentation, a given set of parameter value θ is called a **configuration**.

The rest of this section introduces the distance between observations and the predictions of the model, in Section XI.1.1 while the methods are discussed in their own sections. The already predefined calibration methods proposed in the Uranie platform are listed below:

- The minimisation, discussed in Section XI.3
- The linear Bayesian estimation, discussed in Section XI.4
- The ABC approaches, discussed in Section XI.5
- The Markov-chain Monte-Carlo sampling, discussed in Section XI.6

As for other modules, there is a specific class organisation that links the main classes in this module. The class hierarchy is shown in Figure XI.1 and is discussed a bit here to explain the the two main classes from which everything other classes are derived and corresponding shared function throughout the method. One can see this organisation with the two sets of classes: those inheriting from the `TCalibration` class and those inheriting from `TDistanceFunction` class. The former are the different methods that have been developed to calibrate a model with respect to the observations and each and every method will be discussed in the upcoming sections. Whatever the method under consideration, it always includes a distance function object, which belongs to the latter category and its main job is to quantify how close the model predictions are to the observations. These objects are discussed in the rest of this introduction, see for instance in Section XI.1.1.

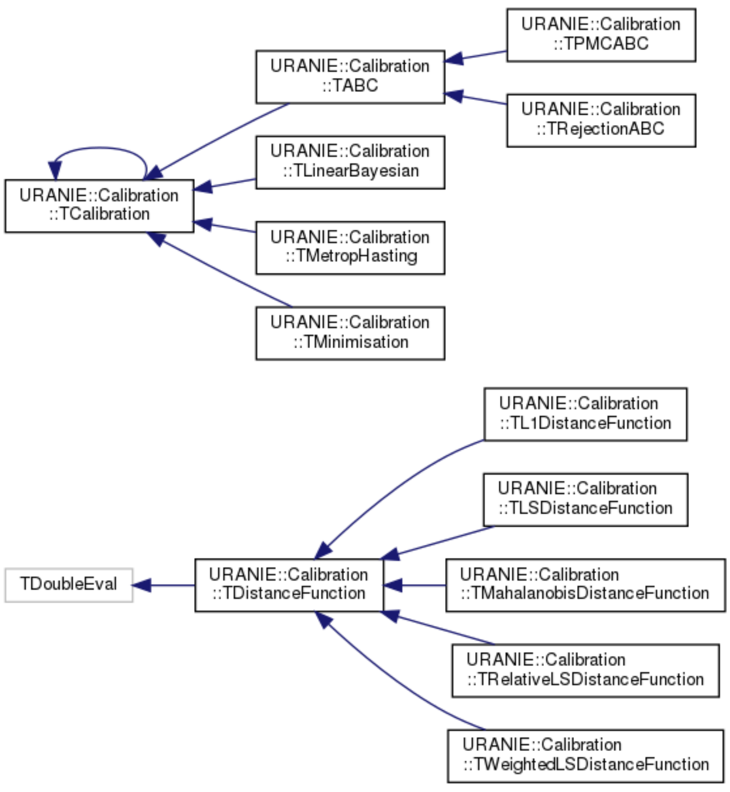

URANIE::Calibration ::TPMCABC
URANIE::Calibration ::TABC
URANIE::Calibration ::TRejectionABC
URANIE::Calibration ::TLinearBayesian
URANIE::Calibration ::TCalibration
URANIE::Calibration ::TMetropHasting
URANIE::Calibration ::TMinimisation
URANIE::Calibration ::TL1DistanceFunction
URANIE::Calibration ::TLSDistanceFunction
TDoubleEval
URANIE::Calibration ::TDistanceFunction
URANIE::Calibration ::TMahalanobisDistanceFunction
URANIE::Calibration ::TRelativeLSDistanceFunction
URANIE::Calibration ::TWeightedLSDistanceFunction

Figure XI.1: Hierarchy of classes and structures out of Doxygen for the `Calibration` module

XI.1.1 The distance used to compare observations and model predictions

There are many ways to quantify the agreement of the observations (our references) with the predictions of the model given a provided vector of parameter θ. As a reminder, this step has to be run every time a new vector of parameter θ is under investigation which means that the code (or function) should be run n times for each new parameter vector.

Starting from the formalism introduced above, many different distance functions can be computed. Given the fact that the number of variable n_Y used to perform the calibration can be different than 1, one might also need variable weight $\{\omega_j\}_{j\in[1,n_Y]}$ that might be used to ponderate the contribution of every variable with respect to the others. Given this, here is a non-exhaustive list of distance functions:

- L1 distance function (sometimes called Manhattan distance): $d(\mathbf{y},\mathbf{f}_\theta(\mathbf{x})) = \sum_{j=1}^{n_Y} \omega_j \times \left(\sum_{i=1}^{n} |\mathbf{y}_i^j - \mathbf{f}_\theta(\mathbf{x})_i^j| \right)$

- Least square distance function: $d(\mathbf{y},\mathbf{f}_\theta(\mathbf{x})) = \sum_{j=1}^{n_Y} \times \sqrt{\omega_j \sum_{i=1}^{n} (\mathbf{y}_i^j - \mathbf{f}_\theta(\mathbf{x})_i^j)^2}$

- Relative least square distance function: $d(\mathbf{y},\mathbf{f}_\theta(\mathbf{x})) = \sum_{j=1}^{n_Y} \times \sqrt{\omega_j \sum_{i=1}^{n} \left(\frac{\mathbf{y}_i^j - \mathbf{f}_\theta(\mathbf{x})_i^j}{\mathbf{y}_i^j} \right)^2}$

- Weighted least square distance function: $d(\mathbf{y},\mathbf{f}_\theta(\mathbf{x})) = \sum_{j=1}^{n_Y} \times \sqrt{\omega_j \sum_{i=1}^{n} \psi_i \times (\mathbf{y}_i^j - \mathbf{f}_\theta(\mathbf{x})_i^j)^2}$ where $\{\psi_i\}_{i\in[1,n]}$ are weights used to ponderate each and every observations with respect to the others.
- Mahalanobis distance function: $d(\mathbf{y},\mathbf{f}_\theta(\mathbf{x})) = \sum_{j=1}^{n_Y} \times \sqrt{\omega_j(\mathbf{y}^j - \mathbf{f}_\theta(\mathbf{x})^j)^T\Sigma^{-1}(\mathbf{y}^j - \mathbf{f}_\theta(\mathbf{x})^j)}$ where Σ is the covariance matrix of the observations.

Their implementation is discussed in Section XI.2.2

XI.2 Calibration classes, distance functions, observations and model

This section introduces the common part of all analysis in the **Calibration** module. Indeed the methods discussed hereafter will be using the same architecture and will be needing a common list of items listed here:

- the model have to be settled as this is what one wants to calibrate. It can comes either as a `Relauncher::TRun` instance, as a `Launcher::TCode` or a **Launcher**'s function. This part is introduced in Section XI.2.1 (for the general concept and the difference with the usual organisation of model definition) and discussed later-on (mainly for the `TCalibration`-inheriting object constructor) in Section XI.2.3 and in the dedicated section in each method.
- the reference observations have to be defined once and for all so that the model can be run for every newly define set of parameter's value (every new configuration). This part is discussed first in Section XI.2.1 and the way to provide them is also partly discussed in Section XI.2.2.
- a distance function as to be created, usually within the calibration instance, to be able to quantify how close the model under study can mimic a set of reference observations. This part is discussed in Section XI.2.2.
- a main object has to be created, a calibration method instance, that inherits from the `TCalibration` class. This is discussed in Section XI.2.3.

XI.2.1 General introduction on data and model definition

All calibration problem will have, at least, two `TDataServer` objects:

- The reference one, usually called `tdsRef`, which contains the observations (both input and output attributes) onto which the calibration is about to be performed. It is generally read from a simple input file as done below:

```
TDataServer *tdsRef = new TDataServer("reference","my reference");
tdsRef->fileDataRead("myInputData.dat");
```

- the parameter one, usually called `tdsPar`, it contains only attributes and must be empty of data. Its purpose is to define the parameters that should be tested in the calibration process and, depending on the method chosen, will only contain `TAttribute` members (for minimisation, see Section XI.3) or only `TStochasticAttribute` inheriting objects for all other methods. The latter case gathers the method doing the analytical computation when the chosen *priors* are allowed (see Section XI.4) along with all those that require generating one or more design-of-experiments (see Section XI.5 but also Section XI.6).

This step, which should represent the first lines of the calibration procedure, goes along with the model definition. This one is tricky with respect to all the examples provided in Section XIV.4 and in Section XIV.8 as the inputs of the model are coming from two different `TDataServers`, they can be split into to two categories:

- the reference ones will only have values from the reference input file `myInputData.dat`, meaning that for every configuration, a reference attribute will take the n predefined values.
- the parameter ones whose value will be changed every time a new configuration will be tested: this value is constant for all the entry of the reference datasets.

Depending on the way the model is coded (and more likely on the parameters the user would like to calibrate) these attributes might not be separated in term of orders, meaning that the list of inputs of a model might look a bit like this:

```
// Example of input list for a fictive model (whatever the launching solution is chosen)
// ref_var1, ref_var2, ref_var3, ref_var4 are coming from the tdsRef dataserver
// par1, par2 are coming from the tdsPar dataserver
TString sinputList="ref_var1:par_1:ref_var2:ref_var3:ref_var4:par_2";
```

Warning
As a result, there is no implicit declaration allowed in the calibration classes constructor and a particular attention must be taken when defining the model: the user must provide the list of inputs (for **Launcher**-type model) or fill the input and output list into the `TEval`-inheriting object in the correct order (for **Relauncher**-type model). This is further discussed in Section XI.2.3.

Finally, all model considered for calibration should have exactly as many outputs (whatever their name are) than the number of output to be compared with (the output attributes in the `tdsRef TDataServer` object). These outputs are those that will be used to compute the chosen agreement (meaning the result of the distance function) which is the only quantifiable measurement we have between the reference and the predictions for a given configuration). At the end of a calibration process, the user can found three different kinds of information (more can be added if needed, see Section XI.2.3.4)

- the resulting parameter's value (or values depending on the chosen method, as some of them are providing several configurations) are being stored in the parameter `TDataServer` object: as this object was provided empty and should contain only an attribute per parameter to be calibrated, it seems to be the best place to store results. This is obviously the expected target but it should not be considered conclusive without having a look at the two other ones
- the agreement between the reference data and the model predictions, which are stored in the parameter `TDataServer` object `tdsPar` for every kept configuration;
- the residues: the difference between the model predictions and the reference data for every n, using the *a priori* and *a posteriori* configuration. These are stored in a dedicated `TDataServer` object, called the **EvaluationTDS** (referred to as `tdsEval`) and it is mainly called through the `drawResidues` method (discussed in Section XI.2.3.7). If one wants to access it, it is possible to get a hand on it by calling the `getEvaluationTDS()` method. The residues are important to check that the behaviour our the newly performed calibration does not show unexpected and unexplained tendency with respect to any variables in the defined uncertainty setup, see the dedicated discussion on this in [30]

XI.2.2 Defining data and distance functions

The different distance function already embedded in Uranie can be found in Section XI.1.1 and are further discussed, from a theoretical point of view in [30]. From the user point of view, on the other hand, every distance function is inheriting from the class `TDistanceFunction`, as can be seen in Figure XI.1, which is purely virtual (meaning that no object can be created as an instance of `TDistanceFunction`) and which deals with several main purposes:

- it loads the reference data once and store them in internally as a vector of vector (or as a vector of `TMatrixD` depending on the chosen formalism used to compute the distance itself);
- once done, the following loop will be called as long as one needs to test new configuration (by configuration we call a new set of values for the θ vector):
  1. it runs the chosen model (disregarding the nature of the object: `TRun`, `TCode`...) on the full reference datasets to get the new predictions.
  2. it loads the new model predictions for the configuration under study into a vector of vector (or as a vector of `TMatrixD` depending on the chosen formalism).
  3. it computes the distance using both vectors as stated by the equations in Section XI.1.1. This computation is done within the `localeval` method (which is the only method that should be redefined if a user wants to create its own distance function, see dedicated discussion in Section XI.2.2.2).

On a technical point of view, the `TDistanceFunction` inherits from the `TDoubleEval` class which is a part of the **Relauncher** module. This inheritance is not very important as its main appeal is to **considerably** simplify the implementation of the minimisation methods with the **Reoptimizer** module, allowing to simply use all **TNlopt** algorithms but also the **Vizir** solutions (see Section XI.3).

Disregarding the considered calibration method, a distance function must be used to compare data and model predictions. This is true even for the `TLinearBayesian` class which only computes the analytical posterior distribution as the residues are computed both *a priori* and *a posteriori* in order to see the improvement of the prediction and possibly their consistency within the uncertainty model (see discussion in [30]). In most of the case (if not all) the object will be constructed with the recommended way (discussed in Section XI.2.2.1). Another possibility is anyway discussed in Section XI.2.2.2

Whatever the situation (either discussed in Section XI.2.2.1 or in Section XI.2.2.2), once a calibration instance is created (for the sake of genericity we will use here an instance, named `cal`, of the fake class `TCalClass` as if it were inheriting from the `TCalibration` class), the first method to be called is the `setDistanceAndReference`, as this is the method with which one defines both the type of distance function and the observation ensemble. The former is further discussed in this section and the latter is crucial as without observations no calibration can be done.

XI.2.2.1 Recommended distance function construction method

The recommended way to create a distance function is to call a method implemented in the `TCalibration` class, which is inherited in every calibration method classes. This method, is called `setDistanceAndReference` and the prototype we're discussing here is the following:

```
void setDistanceAndReference(const char *funcName, TDataServer *tdsRef, const char *input, ↩
    const char *reference, const char *weight="");
```

It takes up to five elements which are:

1. **funcname**: the name of the distance function describes of the already implemented one, as discussed in Section XI.1.1. The possibility are
   - "L1" for `TL1DistanceFunction`
   - "LS" for `TLSDistanceFunction`
   - "RelativeLS" for `TRelativeLSDistanceFunction`
   - "WeightedLS" for `TWeightedLSDistanceFunction`
   - "Mahalanobis" for `TMahalanobisDistanceFunction`

2. **tdsRef**: the `TDataServer` in which the observations are stored;
3. **input**: the input variables stored in the `TDataServer` tdsRef which have been defined as inputs in the code just before creating the calibration object. This argument has the usual attribute list format "x:y:z".
4. **reference**: the reference variables stored in the `TDataServer` tdsRef, with which the output of the code or function will be compared to. This argument has the usual attribute list format "out1:out2:out3".
5. **weight**: this argument is optional and can be used to define the name of the (single) variable stored in the `TDataServer` tdsRef which should be used, in the case of a `TWeightedLSDistanceFunction`, to fill the $\{\psi_i\}_{i\in[1,n]}$, *i.e.* the weights used to ponderate each and every observations with respect to the others (see Section XI.1.1).

Warning A word of cautious about the string to be passed: the number of variable in the list `weight` should match the number of output of your code that you are using to calibrate your parameters. Even in the peculiar case where you'll be doing calibration with two outputs, one being free of weights, then one should add a "one" attribute to provide for the peculiar output if the other one needs uncertainty model.

Once this method is called, the distance function is created and is stored within the calibration object. It might be needed to put a hand on it for some option, but this is further discussed in Section XI.2.2.3.

The following line summarise this construction in a case where an instance `cal` of the fake class `TCalClass` (as if this class were inheriting from the `TCalibration` class) is created.

```
// Define the dataservers
TDataServer *tdsRef = new TDataServer("reference","myReferenceData");
// Load the data, both inputs (ref_var1 and ref_var2) and a single output (ref_out1).
tdsRef->fileDataRead("myInputData.dat");
...
TDataServer *tdsPar = new TDataServer("parameters","myParameters");
tdsPar->addAttribute( new TNormalDistribution("par1",0,1) );
// the parameter to calibrate
...
// Define the model
...
// Create the instance of TCalClass:
TCalClass *cal = new TCalClass(); // Constructor is discussed later-on
// Define the Least-Square distance
cal->setDistanceAndReference("LS", tdsRef, "ref_var1:ref_var2", "ref_out1");
```

In this fake example, the distance function is the Least-square one, and it will used the n values of both inputs `"ref_var1"` and `"ref_var1"` and output `"ref_out1"` stored in tdsRef to calibrate the parameter `"par1"`. No observation weight is needed in this case, as least-square does not require it and as there is only a single output, no variable weight has to be defined as well.

XI.2.2.2 Creating its own distance function

It is possible, if one wants to use a distance function not already implemented in Uranie to ???

XI.2.2.3 Options usable for every distance function

All distance function classes inherit from the `TDistanceFunction` one and the only difference between all of them is the implementation of the `localeval` function. This means that a very large fraction of the code is shared by all the distance function object, including the part that deals with their configurations and options.

This part is gathering all the share options that can be configured either through the optional "Option" in several method and constructors, or by accessing the distance function object itself once created. In order to do so, one should call the `getDistanceFunction` method that will return (if it has been created) the distance function instance stored in the calibration instance under consideration. The following lines provide an example in the case where one is dealing with and instance `cal`, of the fake class `TCalClass` (as if it were inheriting from the `TCalibration` class)

```
// Creating the calibration instance from a TDataServer and a Relauncher::TRun
TCalClass *cal = new TCalClass(tdsPar,runner,1,"");
// Creating the distance function (a least square one)
cal->setDistanceAndReference("LS",tdsRef,"logRe:logPr","logNu");
// Retrieving the instance to be able to change options
TLSDistanceFunction *dFunc = (TLSDistanceFunction*)cal->getDistanceFunction();
```

In the last line of the previous code bloc, the right hand side of the equality started by a cast of the pointer that is returned by `getDistanceFunction`. As this method returns a pointer to a `TDistanceFunction`, in C++ one should precise that here, this object is indeed an instance of `TLSDistanceFunction.`. In the case of python, this is not needed and the equivalent code bloc should look like this:

XI.2.2.3.1 Define the variable weights

In the case where there are several variables used to compare predictions and observations, one might want to ponderate their contribution to the distance. To do two methods are available, both called `setVarWeights`. Their difference is only the prototype :

```
// Prototype 1
void setVarWeights( int nwei, double *wei );
// Prototype 2
void setVarWeights( vector<double> wei);
```

The idea behind this two prototypes is to have a way to control the number of elements in the array of double, either by asking the user to provide it (in prototype 1) or as a by-product of the vector structure. From there, if the size of the array is matching the number of output variables the weights are initialised, the program will crash otherwise.

XI.2.2.3.2 Dump all the estimations

Whatever the construction and evaluation-type (either based on **Launcher** or **Relauncher**), you might want to keep track of all evaluations and not only the global distance for all the reference observations. This is possible by calling the `TDistanceFunction`'s method:

```
void dumpAllDataservers();
```

This methods takes no argument and sets a boolean to true (by default it is set to false) which implies that every configurations will be dumped as an ASCII file that would be named following this convention: `Calibration_testnumber_XX.dat` where `XX` is the configuration number for the under-study analysis.

Warning Two words of caution:

- this options might dump a very large number of files which can fill your local disk (if you're testing functions for instance for which the limit on the number of configuration is not really a limit).
- used with a runner architecture (**Relauncher**-type evaluator), the parameters are not kept by default in the dataserver (they're defined as `ConstantAttribute`). To get the required behaviour (meaning having parameter's value in the ASCI file), the user should also call a `TDistanceFunction`'s method called

  ```
  void keepParametersValue();
  ```

  This function, which takes no argument, just set to true the final argument of all `TLauncher2`'s call of `addConstantAttribute`.

XI.2.2.3.3 Define the observations covariance matrix

In the case where the observations are correlated with a known covariance structure, this covariance can be precised as a `TMatrixD` using the following method:

```
void setObservationCovarianceMatrix(TMatrixD &mat);
```

This method and the possibility to define a covariance structure between the observations is only relevant when the distance function is an instance of the `TMahalanobisDistanceFunction`.

XI.2.2.3.4 Specify a Launcher object or options

This method is very specific as it can be used for all calibration classes, but only if the model introduced is a `Launcher::TCode`. This option pops up here, because the distance function is the place where the model is used to estimate the new predictions for a newly set parameter vector (see Section XI.2.2). This option allows not to use the usual *TLauncher* object in order to use one of the other instance. This is done in a single function:

```
void changeLauncher(TString tlcName);
```

The only argument of this method is a `TString` object which is used to used of the two other instance in the **Launcher** module for the `TCode`: either the `TLauncherByStep` or the `TLauncherByStepRemote`. When used, the `setDrawProgressBar` method is also called to set this variable to `false`.

The main point of this is to switch from an usual `TLauncher` object to another one chosen by the user. Among the possible solutions one can set

- `TLauncherByStep`: if the user wants to breakdown the launching process, decomposing the into `preTreatment`, `run` and `postTreatment`.
- `TLauncherByStepRemote`: if the user wants to use the code on a cluster on which Uranie is not installed. This is not recommended for any cluster set-up as all the CEA clusters are. This is based on the `libssh` library, see Section IV.4.5 for more details.

Finally, the default option for the `run` method of whatever kind of `TLauncher`-inheriting instance is `noIntermediateSteps`, that prevents the safety writing of the results into a file every five estimations. Two methods can be applied on a `TDistanceFunction` objects to change these options:

```
// Adding more options to the code launcher run method
void addCodeLauncherOpt(TString opt);
// Change the code launcher run method options
void changeCodeLauncherOpt(TString opt);
```

The first one is written to keep the default and add more options on top, for instance options that would allow to distribute the computation with the fork process (as a reminder, this option is **"localhost=X"** where X stands for the number of threads to be used). On the other hand, the second method above allow to restart from scratch in order to define the options has chosen by the user.

XI.2.2.3.5 Dump the distance details

In the case where one do not understand or trust the way the distance are computed, it is possible to dump the details of their computation (warning this is really verbose). This can be done by calling the `dumpDetails`, whose prototype is pretty simple:

```
void dumpDetails();
```

XI.2.3 The calibration classes common methods

All the calibration classes that derive from `TCalibration` share a lot of common methods and their organisation has been factorised as much as possible. This section will describe all this shared code, preventing repetition in the upcoming sections that will deal with every specific calibration method (from Section XI.3 to Section XI.6).

From Section XI.2.3.1 to Section XI.2.3.3, one discusses the different constructor explaining the different way the model could be provided and what it implies while Section XI.2.3.4 provides a glimpse of the shared possible options that can be defined and their purpose. Finally Section XI.2.3.6 and Section XI.2.3.7 introduce the drawing method in their principle (but illustration for them will be postponed to the dedicated method in the rest of this documentation).

Warning As for some of the discussions above, the following methods are common to all `TCalibration`-inheriting classes, so the example provided will be written using an instance `cal` of the fake class `TCalClass` (as if this class were inheriting from the `TCalibration` class). A more realistic syntax should be found in the dedicated sections (from Section XI.3 to Section XI.6).

The calibration classes can, generally, be constructed with 4 different ways, linked to the way the model has been precised. To estimate how close the new set of parameter values (the **configuration**) is to the reference data, one needs to be able to run the model on the data's input variable. The input variables of the observation datasets are not the only input variable of the model used within the calibration method, as the parameters themselves have to be specified as inputs (as they also obviously affect the predictions). This section shows, in the fictive case of a **not existing** `TCalClass` class how to construct our calibration objects.

XI.2.3.1 Construction with a runner

This constructor is using the **Relauncher** architecture. This approach allows a simple way to change the evaluator (to pass from a C++ function to a python's one or a code) but also to use either a sequential approach (for a code) to a threaded one (to distribute locally the estimations). This approach is partly discussed in Chapter VIII.

The constructor in this case, should look like this

```
// Constructor with a runner
TCalClass(TDataServer *tds, TRun *runner, Int_t ns=1, Option_t *option="")
```

It takes up to four elements which are:

1. **tds**: a `TDataServer` object containing only an attribute for every parameter to be calibrated. This is the `TDataServer` object called `tdsPar`, defined in Section XI.2.1.
2. **runner**: a `TRun`-inheriting instance that contains all the model information and whose type is defining the way to distribute the estimation: it can either be a `TSequentialRun` instance or `TThreadedRun` for distributed computations.
3. **ns**: the number of samples to be produced. This field only applies to methods for which more than one configuration are expected which is not the case for local minimisation with a single point initialisation but also for linear Bayesian analysis (see Section XI.4). The **default** value is 1.
4. **option**: the option that can be applied to the method. The option common to all calibration classes (so those defined in the `TCalibration` class) are discussed in Section XI.2.3.4.

The key step in this constructor is the `TRun`-inheriting instance creation. As already stated, its type is giving the lead on the way to distribute the estimations. When one is constructing such an object, it is done by passing an evaluator, whose list is already largely discussed in Section VIII.3.

Taking back the formalism already introduced in Section XI.2.2.1, the model instance of a function `Foo`, a classical `TCIntEval` function, is created as done below, when this model takes the following inputs `ref_var1`, `par1` and `ref_var2` and it produce a single output to be compared with `"ref_out1"` (the comparison between this reference and the model prediction is done through the distance function, as already discussed Section XI.2.2.1 in for this example).

```
// Define the dataservers
TDataServer *tdsRef = new TDataServer("reference","myReferenceData");
// Load the data, both inputs (ref_var1 and ref_var2) and a single output (ref_out1).
tdsRef->fileDataRead("myInputData.dat");
...
TDataServer *tdsPar = new TDataServer("parameters","myParameters");
tdsPar->addAttribute( new TNormalDistribution("par1",0,1) ); // the parameter to calibrate
...

// Define the model if a function Foo(double *x, double *y)  is defined above
// for which x[0]=ref_var1, x[1]=par1 and x[2]=ref_var2 and y[0] is the model prediction
TCIntEval *model = new TCIntEval("Foo");
// Add inputs in the correct order
model->addInput(tdsRef->getAttribute("ref_var1"));
model->addInput(tdsPar->getAttribute("par1"));
model->addInput(tdsRef->getAttribute("ref_var2"));
// Define the output attribute
TAttribute *out = new TAttribute("out");
model->addOutput(out);
// Define a sequential runner to be used
TSequentialRun *runner = new TSequentialRun(model);

...
// Create the instance of TCalClass:
int ns=1;
TCalClass *cal = new TCalClass(tdsPar, runner, ns, "");
```

XI.2.3.2 Construction with a `TCode`

This constructor is using the **Launcher** architecture. This approach is pretty different from the **Relauncher** one, as it is only focusing on the code case.

The constructor in this case, should look like this

```
// Constructor with a TCode
TCalClass(TDataServer *tds, TCode *code, Int_t ns=1, Option_t *option="")
```

It takes up to four elements which are:

1. **tds**: a `TDataServer` object containing only an attribute for every parameter to be calibrated. This is the `TDataServer` object called `tdsPar`, defined in Section XI.2.1.
2. **code**: a `TCode` instance containing the output file (or files) that list the output attributes while all input attributes have been assigned an input file by the usual methods (`setFileKey`, `setFileFlag`...).
3. **ns**: the number of samples to be produced. This field only applies to methods for which more than one configuration are expected which is not the case for local minimisation with a single point initialisation but also for linear Bayesian analysis (see Section XI.4). The **default** value is 1.
4. **option**: the option that can be applied to the method. The option common to all calibration classes (so those defined in the `TCalibration` class) are discussed in Section XI.2.3.4.

Unlike the runner constructor discussed above, this construction does not bring any information on the way the computation will be performed. The `run` method is called for every configuration with the following options as a default: **noIntermediateSteps**, that prevents the safety writing of the results into a file every five estimations and **quiet** that prevents the launcher to be too verbose. Two methods can be applied on a `TDistanceFunction` objects to change these options:

```
// Adding more options to the code launcher
void addCodeLauncherOpt(TString opt);
// Change the code launcher option
void changeCodeLauncherOpt(TString opt);
```

The first one is written to keep the default and add more options on top, for instance options that would allow to distribute the computation with the fork process (as a reminder, this option is **"localhost=X"** where X stands for the number of threads to be used). On the other hand, the second method above allow to restart from scratch in order to define the options has chosen by the user.

Taking back the formalism already introduced in Section XI.2.2.1, the model instance of a code `Foo`, using a `TCode` instance, is shown below, when this model takes the following inputs `ref_var1`, `par1` and `ref_var2` and it produces a single output to be compared with `"ref_out1"` (the comparison between this reference and the model prediction is done through the distance function, as already discussed Section XI.2.2.1 in for this example).

```
// Define the dataservers
TDataServer *tdsRef = new TDataServer("reference","myReferenceData");
// Load the data, both inputs (ref_var1 and ref_var2) and a single output (ref_out1).
tdsRef->fileDataRead("myInputData.dat");
...
TDataServer *tdsPar = new TDataServer("parameters","myParameters");
tdsPar->addAttribute( new TNormalDistribution("par1",0,1) ); // the parameter to calibrate
...

TString sIn = TString("code_foo_input.in");
// Set the reference input file and the key for each input attributes
```

```
tdsRef->getAttribute("ref_var1")->setFileKey(sIn, "var1");
tdsRef->getAttribute("ref_var2")->setFileKey(sIn, "var2");
tdsPar->getAttribute("par1")->setFileKey(sIn, "par1");
// The output file of the code
TOutputFileRow *fout = new TOutputFileRow("_output_code_foo_.dat");
// The attribute in the output file
fout->addAttribute(new TAttribute("out"));
// Creation of the code
TCode *code = new TCode(tdsRef, "foo -s -k");
mycode->addOutputFile( fout );

...
// Create the instance of TCalClass:
int ns=1;
TCalClass *cal = new TCalClass(tdsPar, code, ns, "");
```

XI.2.3.3 Construction for a function with the Launcher architecture

This constructor is using the **Launcher** architecture and deals with a function (C++ one with the usual prototype).

The constructors in this case, should look like this

```
// Constructor with a function pointer using Launcher
TCalClass(TDataServer *tds, void (*fcn)(Double_t*,Double_t*), const char *varexpinput,  ↩
    const char *varexpoutput, int ns=1, Option_t *option="");
// Constructor with a function name using Launcher
TCalClass(TDataServer *tds, const char *fcn, const char *varexpinput, const char * ↩
    varexpoutput, int ns=1, Option_t *option="");
```

It takes up to six elements, four of which are compulsory:

1. **tds**: a `TDataServer` object containing only an attribute for every parameter to be calibrated. This is the `TDataServer` object called `tdsPar`, defined in Section XI.2.1.
2. **fcn**: the second argument is either the name of the function or a pointer to this function. A good knowledge of this function implies that the user must know in which order the input and output variables are provided.
3. **varexinput**: the list of input variables in the correct order, as an admixture of the reference attributes and the parameter attributes.
4. **varexpoutput**: the list of output variables.
5. **ns**: the number of samples to be produced. This field only applies to methods for which more than one configuration are expected which is not the case for local minimisation with a single point initialisation but also for linear Bayesian analysis (see Section XI.4). The **default** value is 1.
6. **option**: the option that can be applied to the method. The option common to all calibration classes (so those defined in the `TCalibration` class) are discussed in Section XI.2.3.4.

This constructor is the simplest one, as all information are provided on a single line, no option has to be defined and no file should be created. Taking back the formalism already introduced in Section XI.2.2.1, the model being considered is the function `Foo`, and the construction using the pointer prototype is shown below, when this model takes the following inputs `ref_var1`, `par1` and `ref_var2` and it produces a single output to be compared with `"ref_out1"` (the comparison between this reference and the model prediction is done through the distance function, as already discussed Section XI.2.2.1 in for this example).

```
// Define the dataservers
TDataServer *tdsRef = new TDataServer("reference","myReferenceData");
// Load the data, both inputs (ref_var1 and ref_var2) and a single output (ref_out1).
tdsRef->fileDataRead("myInputData.dat");
...
TDataServer *tdsPar = new TDataServer("parameters","myParameters");
tdsPar->addAttribute( new TNormalDistribution("par1",0,1) ); // the parameter to calibrate
...

// Create the instance of TCalClass:
int ns=1;
TCalClass *cal = new TCalClass(tdsPar, Foo, "ref_var1:par1:ref_var2", "out", ns, "");
```

XI.2.3.4 Running the estimation

Once the calibration-object is constructed and its distance function is also created, the aim is to perform the calibration, meaning that the best value (or values) of the parameters have to be estimated. This is done by calling the method

```
void estimateParameters(Option_t *option="");
```

This method is global and it mainly call another internal method, defined in every `TCalibration`-inheriting class, in which the real estimation is performed.

There are different options that can be applied to the `estimateParameters` method, among which:

"saveAllEval" this option allows to keep every single estimations in the internal dataserver that is later-on used to produce residue plots (see Section XI.2.3.7). WARNING: this can very likely become unbearable, as the number of estimation to keep might be gigantic.

"noAgreement" this options allows to get rid of the *agreement* attribute at the end of the estimation if one considers the information is pointless.

Also, some options, not discussed here because they're triggered by calling methods on the `TDistanceFunction`-inheriting instance, might be of use to understand and validate the way the calibration has been performed. For this, see Section XI.2.2.3.

There are few methods used to represent the results. The skeleton are defined within the `TCalibration` class (even though some might not applied to few specific method). This section introduces the general concept and every calibration method particularity will be discussed within the proper method section.

XI.2.3.5 Estimate custom residues

Once the estimation is performed, the *a priori* and *a posteriori* residuals are estimated, meaning that from both set of parameter values (the *a priori* and *a posteriori* values estimation will depend on the chosen algorithm so this is not specified here) one can ask to re-evaluate the residuals once another set of parameter values is provided. This can be done through

```
void estimateCustomResidues(string resName, int theta_nb, double *theta_val);
```

which takes three arguments:

resName the name of the set to be kept as a tag;

theta_nb the number of values provided in the array (must be coherent with `_nPar` obviously);

theta_val the value of the parameters to be used as an array.

This construction should be called simply with a vector<double>numpy array for the sake of simplicity as shown below.

```
vector<double> mypar = {0., 2., 3.};
mycal->estimateCustomResidues("set1", mypar.size(), &mypar[0]);
```

The idea behind all this is to be able to re-estimate residuals when the *a posteriori* values of the parameter might have to be carefully estimated provided the sample available in after the estimation (for instance in the case of Markov-Chain to get rid of the warming-up estimations). Once done, the selection can be used in the `drawResidues` function discussed later-on in Section XI.2.3.7.

XI.2.3.6 Drawing the parameters

This method's purpose is to draw parameter's value. The prototype is the following

```
void drawParameters(TString sTitre, const char *variable = "*", const char *select = "1>0", ↩
     Option_t * option = "");
```

It takes up to four arguments, three of which are optional:

sTitre The title of the plot to be produced (an empty string can be put).

variable This field should contain the parameter list to be drawn (with the usual format of parameter's name splitted by ":"). The default value is to draw all parameters (the `"*"` field).

select A selection field to remove some kept configurations (for instance if you want to consider the *burn-in* period or lag procedure for the Metropolis-Hasting algorithm, see Section XI.6).

option The optional field can be used to tune a bit the plots, options being separated by commas

- "nonewcanvas": if this option is used the plot will be done in an existing canvas, if not, a new `TCanvas` will be created on the spot.
- "vertical": if this option is used and more than one parameters have to be displayed, the canvas is splitted into as many windows as parameters in the variable list, the windows being stacked one above the others, using the full width. The default is "horizontal" so the plots are side-by-side.

XI.2.3.7 Drawing the residues

This method's purpose is to draw output variable's residue, meaning the difference between the model predictions and the reference output. Two kinds of residues are generally discussed: the *a priori* ones (from the initialisation values of the parameters) and the *a posteriori* ones (the results of the calibration procedure). The prototype is the following

```
void drawResidues(TString sTitre, const char *variable = "*", const char *select="1>0",  ↩
    Option_t * option = "");
```

It takes up to four arguments, three of which are optional:

sTitre The title of the plot to be produced (an empty string can be put).

variable This field should contain the output variable list to be drawn (with the usual format of parameter's name splitted by ":"). The default value is to draw all variables (the `"*"` field).

select A selection field to remove some of the observations from the reference datasets.

option The optional field can be used to tune a bit the plots, options being separated by commas

- "nonewcanvas": if this option is used the plot will be done in an existing canvas, if not, a new `TCanvas` will be created on the spot.
- "vertical": if this option is used and more than one parameters have to be displayed, the canvas is splitted into as many windows as parameters in the variable list, the windows being stacked one above the others, using the full width. The default is "horizontal" so the plots are side-by-side.
- "apriori/aposteriori": these options state whether only the *a priori* residues (with "apriori") or only the *a posteriori* (with "aposteriori") field. If none of this two fields is used, both kind of residues should be drawn.
- "custom=XXX": this option state whether one also wants to show the custom residues estimate through the `esimateCustomResidues` method already introduced in Section XI.2.3.5. With the example provided in the aforementionned section, the command can look like this, if one wants to compare *a priori*, *a posteriori* and the "set1" custom set:

```
mycal->drawResidues("Residual title", "*", "", "nonewcanvas,apriori,aposteriori, ↩
    custom=sel1");
```

Summary: `TCalibration` and `TDistanceFunction`

1. Define both the parameter (`tdsPar`) and reference (`tdsRef`) objects, as explained in Section XI.2.1.
2. Define the model architecture (either **Relauncher** or **Launcher**-based) and the evaluator kind (code, C++-function, python-function...). A peculiar attention is needed to specify the input and output attributes (the former should be an admixture of attributes from `tdsPar` and `tdsRef`, as explained in Section XI.2.1.
3. Define the `TCalibration`-inheriting object (meaning the underlying method chosen for this analysis) and construct it (see from Section XI.2.3.1 to Section XI.2.3.3).
4. Define the `TDistanceFunction`-inheriting object by calling the `TCalibration`'s method `setDistanceAndReference`, see the recommended method in Section XI.2.2.1.
5. Set the method-dependent parameters (discussed in dedicated part, from Section XI.3 to Section XI.6).
6. Call the `estimateParameters` method with selected options (see Section XI.2.3.4).
7. Do the post-treatment thanks to method-dependent functions discussed in dedicated part, from Section XI.3 to Section XI.6) and drawing methods (see also both Section XI.2.3.6 and Section XI.2.3.7).

XI.2.4 Use-case for this chapter

In order to illustrate the ongoing methods in the future sections, a bit more closely than the already introduced dummy examples, a general use-case will be used. This use-case rely on the `flowrate` model introduced (along with a descriptive sketch) in Chapter IV and whose equation is recalled below:

$$y = f(x) = \frac{2\pi T_u (H_u - H_l)}{\ln(\frac{r}{r_\omega})\left[1 + \frac{2LT_u}{\ln(\frac{r}{r_\omega}) r_\omega^2 K_\omega} + \frac{T_u}{T_l}\right]}$$

EQUATION XI.1: Flowrate function

where the eight parameters are:

1. $r_\omega \in [0.05, 0.15]$ (m): radius of borehole
2. $r \in [100, 50\,000]$ (m): radius of influence
3. $T_u \in [63\,070, 115\,600]$ $(m^2/year)$: Transmitivity of the superior layer of water
4. $T_l \in [63.1, 116]$ $(m^2/year)$: Transmitivity of the inferior layer of water
5. $H_u \in [990, 1\,110]$ (m): Potentiometric "head" of the superior layer of water
6. $H_l \in [700, 820]$ (m): Potentiometric "head" of the inferior layer of water
7. $L \in [1\,120, 1\,680]$ (m): length of borehole
8. $K_\omega \in [9\,855, 12\,045]$ (m): hydraulic conductivity of borehole

This example has been treated by several authors in the dedicated literature, for instance in [40]. With respect to our concerns, the idea of the upcoming examples (in this chapter, along with those in the use-case macros section, see Section XIV.11) is to consider that one has an observation sample. For this function, we consider that from all the inputs, only two are have been varied (r_ω and L) and only one is actually unknown: H_l. The rest of the variables are set to a frozen value: $r = 25050$, $T_u = 89335$, $T_l = 89.55$, $H_u = 1050$, $K_\omega = 10950$. This can be written as the following function (using the usual C++ prototype)

```
void flowrateModel(double *x, double *y) {
  double rw = x[1], r = 25050;
  double tu = 89335, tl = 89.55;
  double hu = 1050, hl = x[0];
  double l = x[2], kw = 10950;

  double num = 2.0 * TMath::Pi() * tu * (hu - hl);
  double lnronrw = TMath::Log(r / rw);
  double den = lnronrw * (1.0 + (2.0 * l * tu) / (lnronrw * rw * rw * kw) + tu / tl);

  y[0] = num / den;
}
```

As discussed previously, this function shows that one might be aware of the way the inputs are organised. In this case, the parameter to be calibrated (H_l) comes first while the varying inputs (r_ω and L) come later. The first lines of all examples should look like this

```
// Name of the input reference file
TString ExpData="Ex2DoE_n100_sd1.75.dat";

// define the reference
TDataServer *tdsRef = new TDataServer("tdsRef","doe_exp_Re_Pr");
tdsRef->fileDataRead(ExpData.Data());

// define the parameters
TDataServer *tdsPar = new TDataServer("tdsPar","pouet");
```

```
tdsPar->addAttribute(new TAttribute("hl",700.0,760.0)); // if stochastic laws are needed
// use tdsPar->addAttribute( new TUniformDistribution("hl", 700.0, 760.0) );

// Create the output attribute
TAttribute *out = new TAttribute("out");
```

XI.3 Using minimisation techniques

Warning
This method is fully relying on the **Relauncher** architecture so the only constructor available is the runner one (discussed in Section XI.2.3.1). This means there is no constructor based on `TCode` or function (respectively described in Section XI.2.3.2 and Section XI.2.3.3). This is explained as this method is a using the `Nlopt`-algorithm bank, introduced in Chapter IX but also the **Vizir** package for multi and many criteria algorithm.

Even though the theory behind this method is not revolutionary, these methods are interested and are, historically and conceptually, the simplest one, one can think of. Because of the way it has been organised, it can be used with all **Relauncher** assessors and can call all `NLopt` algorithms along with the `Vizir` ones. This is explained in Section XI.3.2.

XI.3.1 Constructing the instance

As stated above, the only constructor available is the one whose prototype contains an instance of **TRun**-inheriting object. This approach allows a simple way to change the evaluator (to pass from a C++ function to a python's one or a code) but also to use either a sequential approach (for a code) to a threaded one (to distribute locally the estimations).

The constructor in this case, should look like this

```
// Constructor with a runner
TMinimisation(TDataServer *tds, TRun *runner, Int_t ns=1, Option_t *option="");
```

It takes up to four elements which are:

1. **tds**: a `TDataServer` object containing only an attribute for every parameter to be calibrated. This is the `TDataServer` object called `tdsPar`, defined in Section XI.2.1.
2. **runner**: a `TRun`-inheriting instance that contains all the model information and whose type is defining the way to distribute the estimation: it can either be a `TSequentialRun` instance or `TThreadedRun` for distributed computations.
3. **ns**: the number of samples to be produced. This field does not apply here.
4. **option**: the option that can be applied to the method. The option common to all calibration classes (so those defined in the `TCalibration` class) are discussed in Section XI.2.3.4.

The key step in this constructor is the `TRun`-inheriting instance creation. As already stated, its type is giving the lead on the way to distribute the estimations. When one is constructing such an object, it is done by passing an evaluator, whose list is already largely discussed in Section VIII.3.

XI.3.2 Setting the optimisation properties

Once the object is constructed, the optimisation properties must be set, to decide which algorithm must be used. This can be precised by calling the setOptimProperties, whose prototype is the following:

```
// Prototype for nlopt
void setOptimProperties(URANIE::Reoptimizer::TNloptSolver *solv, Option_t *option="");
// Prototype for vizir
void setOptimProperties(URANIE::Reoptimizer::TVizirSolverShare *solv, Option_t *option="");
```

In both cases, the option field is there if some option shows up at some point, even though so far it is useless. There is only one difference between these two prototype: the kind of solver to be used knowing that the minimisation will be done on the distance function value disregarding the number of parameter to calibrate.

Once the optimisation solver is chosen and configured, is provided to the TMinimisation instance, which we'll create automatically the optimisation master depending on the nature of the solver:

TNlopt This is the optimisation master used for any TNloptSolver instance. An example can be found in Section XIV.11.1.

TVizir2 This is the optimisation master used for any TVizirSolverShare instance. Even though the optimisation will remain mono-criterion, this might be useful for some intricate issues, as discussed in Section XIV.11.6.

As shown in the latter example, it is possible to get a hand at the optimisation master by calling the getOptimMaster method. This method returns a pointer to the TOptimShare-newly created instance. This might be useful to set some properties (for instance the tolerance).

Warning Caution has to be taken when calling the getOptimMaster method in C++, as the pointer can be either a TNlopt or TVizir2 instance. If in python this should fully transparent, in C++ it might be useful to cast the pointer returned by this method, as shown below:

```
// Set the calibration object
TMinimisation *cal = new TMinimisation(tdsPar,runner,1);
cal->setDistanceAndReference("relativeLS",tdsRef,"rw:l","Qexp");
bool vizir=true;
if( vizir ){ // Set Viziroptimisaiton properties
  TVizirGenetic solv;
  solv.setSize(24,15000,100);
  cal->setOptimProperties(&solv);
  TVizir2 *optimMaster = ((TVizir2*)cal->getOptimMaster());
} else{ // Set Nlopt optimisaiton properties
  TNloptSubplexe solv;
  cal->setOptimProperties(&solv);
  TNlopt *optimMaster = ((TNlopt*)cal->getOptimMaster());
}
```

Apart from the discussion above, there are no specific options and methods in the TMinimisation class, and more information on method can be found in Section XI.2. Examples are also provided in the use-case sections (see Section XIV.11), particularly one can have a loot at the residue distributions both *a priori* and *a posteriori* for a point-estimation, which are shown in Figure XIV.97.

XI.4 Analytical linear Bayesian estimation

This method is pretty simple from the algorithm point of view as it consists mainly in the analytical formulation of the posterior distribution when the hypotheses on the prior are well set: the problem can be considered linear and the prior distributions are normally distributed (or flat, as discussed in [30]). Practically, handling this technique is done by following the recipe provided in Section XI.2 with an important difference though: the code or function, brought through the constructor of the `TLinearBayesian` object, is not strictly speaking useful. The parameters estimation is indeed analytical so the main point of providing an assessor is to get both the *a priori* and *a posteriori* residue distributions. The important steps of this kind of analysis are gathered here, all classical steps being gathered in the first item:

1. Get the reference data, the model and its parameters. Choose the assessor type you'd like to use and construct the `TLinearBayesian` object accordingly with the suitable distance function. Even though this mainly relies on common code, this part is introduced also in Section XI.4.1, in particular for what happened to the distance function (pay attention to the warning bloc).
2. Provide the input covariance matrix, *i.e.* the reference observation covariance (in [30] this would correspond to the Σ). This is compulsory to get a valid estimation. This impact the distance function choice as discussed in Section XI.4.2.
3. Provide the name of the regressors. This is also a key step as a regressor can be an input variable, but also any function of one or many input variables, This is discussed in Section XI.4.2.
4. A transformation function can be provided but this is not compulsory. This is discussed in Section XI.4.3.
5. Finally the estimation is performed and the results can be extracted or draw with the usual plots. The specificities are discussed in Section XI.4.3.

XI.4.1 Constructing the TLinearBayesian object

The constructors that can be used to get an instance of the `TLinearBayesian` class are those detailed in Section XI.2.3. As a reminder the prototype available are these ones:

```
// Constructor with a runner
TLinearBayesian(TDataServer *tds, TRun *runner, Int_t ns=1, Option_t *option="");
// Constructor with a TCode
TLinearBayesian(TDataServer *tds, TCode *code, Int_t ns=1, Option_t *option="");
// Constructor with a function  using Launcher
TLinearBayesian(TDataServer *tds, void (*fcn)(Double_t*,Double_t*), const char *varexpinput ↩
    , const char *varexpoutput, int ns=1, Option_t *option="");
TLinearBayesian(TDataServer *tds, const char *fcn, const char *varexpinput, const char * ↩
    varexpoutput, int ns=1, Option_t *option="");
```

The details about these constructor can be found in Section XI.2.3.1, Section XI.2.3.2 and Section XI.2.3.3 respectively for the `TRun`, `TCode` and `TLauncherFunction`-based constructor. In all cases, the number of samples ns is set to 1 by default and changing it will not change the results. As for the option, there are no specific options for this class.

The final step here it to construct the `TDistanceFunction` which is the compulsory step which should always come right after the constructor, but a word of caution about this step:

Warning Whatever the distance function you're choosing, the `setDistanceAndReference` is locally redefined so that the distance function will only be a `TMahalanobisDistance`. As defining the observation covariance matrix is mandatory, it would make little sense to use any other distance function which would not use the full extend of the input information. Furthermore, the distance function, in this method, is only provided for illustration purpose, to check the difference between the *a priori* and *a posteriori* parameter's values.

XI.4.2 Define the linear model properties

Once the TLinearBayesian instance is created along with its TDistanceFunction, two methods must be called before getting into the parameters estimation. These methods are compulsory as they will define the heart of the analytical formula to get the Gaussian parameter value of the *a posteriori* distribution (see [30]).

The first one (even though there is no particular order between the two) is setRegressor, whose prototype is

```
void setRegressor(const char *regressorname);
```

The only argument is the regressorname field, which is the list of regressor names split by ":", using the usual format and this method, then, checks two things. The first one is the fact that the number of regressors must match the number of parameters to be calibrated. On top of this, the code passes through the list of attributes available in the reference observation TDataServer and check that every regressor name provided matches one existing attributes. As stated above, if the observation TDataServer does not contains the regressors (when the input file is loaded) these attributes have to be constructed from scratch either through TAttributeFormula or by using another dedicated assessor (as done in the use-case shown in Section XIV.11.2).

The other method is setObservationCovarianceMatrix whose prototype is

```
void setObservationCovarianceMatrix(TMatrixD &mat);
```

The only argument here is a TMatrixD whose content is the covariance matrix of the reference observation data. Once again, this method will check two things:

- the provided matrix must have the correct number of rows and columns (basically both should be set to n);
- the provided matrix should be symmetrical;

Given this, estimations can be performed. One can find an example of how to use these methods in the use-case dedicated subsection, more precisely in Section XIV.11.2.

XI.4.3 Look at the results

Finally once the computation is done there are three different kinds of results and several ways to interpret but also transform it. This section details some important and specific aspect to it.

XI.4.3.1 Transformation of the results

The idea is that when you want to consider your model as linear, you might have to transform it a bit to have a correct linear behaviour and to express and compute the needed regressors. For this peculiar behaviour, one will rely on our use-case discussed in Section XIV.11.2. In this case, one should linearise the flowrate function as done here by writing:

$$f_\theta(x) = (2\pi T_u)\left(\ln(\frac{r}{r_\omega})\left[1+\frac{2LT_u}{\ln(\frac{r}{r_\omega})r_\omega^2 K_\omega}+\frac{T_u}{T_l}\right]\right)^{-1}\theta = H\times\theta$$

where the regressor can be expressed as $H = (2\pi T_u)\left(\ln(\frac{r}{r_\omega})\left[1+\frac{2LT_u}{\ln(\frac{r}{r_\omega})r_\omega^2 K_\omega}+\frac{T_u}{T_l}\right]\right)^{-1}$. From there, it is clear that we will be calibrating a newly defined parameter $\theta = (H_u - H_l)$, so we will have to transform that back into our parameter of interest at some point.

This is the the sole reason why the method `setParameterTransformationFunction` has been implemented: transform the parameters estimated given the linear regressor, the observation covariance matrix and prior distribution. As the transformations, it they do exist which is not compulsory, are expected to be done with simple operations using constant values, they should not affect the covariance matrix of the posterior multidimensional normal distribution, only the mean vector. The prototype of this function is as follows:

```
void setParameterTransformationFunction(void (*fTransfoParam)(double *in, double *out));
```

Its only argument is a pointer to the transformation function in the usual C++ prototype. This function provided to get the proper values of the under-investigation parameters take two arguments: the input parameters which are the raw one estimated from the analytical formula detailed in [30] and the output ones, which should be the ones one wants to have. Both parameters are double-array whose size must be the number of parameters.

The example provided in the use-case is really simple as there is only one parameter to be estimated, which implies that both argument are one-dimension double array which should look like this:

```
void transf(double *x, double *res)
{
    res[0] = 1050 - x[0];  // simply H_l = \theta - H_u
}
```

Warning This is also possible in python but the transformation will still have to be the usual C++ one. To do so, the function has to be put in a C-file, for example called `myFunction.C` and this file has to be loaded in order to get the handle on the function. Here are the few lines that summarise these two steps in a fictional macro that would define a `cal` instance of `TLinearBayesian`:

```
# Define all the needed material: dataservers, models...
cal=Calibration.TLinearBayesian(...) # Create the instance and distance function
# ...
# Load the file in which Transformation function
ROOT.gROOT.LoadMacro("myFunction.C")
# Provide this function to the TLinearBayesian instance
cal.setParameterTransformationFunction(ROOT.transf)
```

XI.4.3.2 Accessing the results

When the estimation is done, it is possible to access the results numerically by calling three methods detailed below. In all cases, the prototype is the same as these functions take no argument and return a `TMatrixD` instance filled with corresponding information. The functions are:

getParameterValueMatrix It returns the raw value of the parameters, meaning the way they have been estimated through the analytical formula. It should return a `TMatrixD` object that should look like a vector (only one-varying dimension).

getParameterCovarianceMatrix It returns the covariance matrix of the estimated parameters, which means that the `TMatrixD` object should be symmetrical and have a $(nPar, nPar)$ dimension.

getTransfParameterValueMatrix It returns the transformed value of the parameters, in case the `setParameterTransfo` has been called properly. It should return a `TMatrixD` object that should look like a vector (only one-varying dimension).

XI.4.3.3 Drawing the parameters

The parameters can be drawn with the newly-defined instance of `drawParameters` whose prototype is the same as the original one discussed in Section XI.2.3.6.

```
void drawParameters(TString sTitre, const char *variable = "*", const char *select = "1>0", ↩
    Option_t * option = "");
```

It takes up to four arguments, three of which are optional:

sTitre The title of the plot to be produced (an empty string can be put).

variable This field should contain the parameter list to be drawn (with the usual format of parameter's name splitted by ":"). The default value is to draw all parameters (the `"*"` field).

select A selection field to remove some kept configurations, which is useless in our case as no events are drawn, only analytical functions, see below.

option The optional field can be used to tune a bit the plots, options being separated by commas

- "nonewcanvas": if this option is used the plot will be done in an existing canvas, if not, a new `TCanvas` will be created on the spot.
- "vertical": if this option is used and more than one parameters have to be displayed, the canvas is splitted into as many windows as parameters in the variable list, the windows being stacked one above the others, using the full width. The default is "horizontal" so the plots are side-by-side.
- "apriori/aposteriori": by default, both distributions are drawn. If this not what's wanted, it is possible to precise either "apriori" or "aposteriori".
- "transformed": if this *posteriori* distribution has to be drawn, this option states that the transformed values should be used as the mean-vector of the multivariate normal posterior distribution.

The main difference with the usual instance of `drawParameters` defined in `TCalibration` is that the object drawn are analytical functions.

On top of the parameters, the residues can also be drawn by calling the `drawResidue` method and no modification has been done to it (for more details, see Section XI.2.3.7).

XI.4.4 Prediction of the variance

Once the estimation has been done, it is obviously possible to estimate the central value for a new set of input values (meaning a new design-of-experiments) thanks to the newly estimated values of the parameters. Even though this is true for every methods in the calibration module, the fact that the LinearBayesian procedure provides the covariance matrix of the parameters means that it is possible (keeping in mind the hypothesis on the input law nature) to get a variance from every newly predicted values that should reflect the uncertainty sorely from the parameters. For more information on the estimation, please have a look at [30]. This can be done by calling the `computePredictionVariance` method:

```
void computePredictionVariance(URANIE::DataServer::TDataServer *tdsPred, string outname);
```

This methods takes two arguments:

tdsPred a dataserver that contains new location to be estimated and **in which all regressors should be available in order to be able to compute the covariance matrix**.

outname the name of the attribute that would be created and which will be filled with the diagonal part (the variance) of the Σ_θ^{pred} matrix.

XI.5 The Approximation Bayesian Computation techniques (ABC)

This sections is discussing methods gathered below the ABC acronym, which stands for *Approximation Bayesian Computation*. The idea behind these methods is to perform Bayesian inference without having to explicitly evaluate the model likelihood function, which is why these methods are also referred to as **likelihood-free** algorithms [21].

As a reminder of what's discussed in further details in [30], the principle of the Bayesian approach is recap in the equation $\pi_{post}(\theta|\mathbf{y}) = \frac{L(\mathbf{y}|\theta)\pi_{prior}(\theta)}{\pi(\mathbf{y})} \propto L(\mathbf{y}|\theta)\pi_{prior}(\theta)$. where $L(\mathbf{y}|\theta)$ represents the conditional probability of the observations knowing the values of θ, $\pi_{prior}(\theta)$ is the *a priori* probability density of θ (the **prior**) and $\pi(\mathbf{y})$ is the marginal likelihood of the observations, which is constant (for more details see [30]).

On the technical point of view, methods in this section will inherit from the TABC class (which itself inherits from the TCalibration one, in order to benefit from all the already introduced features). So far the only ABC method is the Rejection one, discussed in [30] and whose implementation has been done through the TRejectionABC class discussed below.

The way to use our Rejection ABC class is summarised in few key steps here:

1. Get the reference data, the model and its parameters. The parameters to be calibrated must be TStochasticAttri inheriting instances. Choose the assessor type you'd like to use and construct the TRejectionABC object accordingly with the suitable distance function. Even though this mainly relies on common code, this part is introduced also in Section XI.5.1.

2. Provide algorithm properties, to define optional behaviour and precise the uncertainty hypotheses you want, through the methods discussed in Section XI.5.2.

3. Finally the estimation is performed and the results can be extracted or draw with the usual plots. The specificities are discussed in Section XI.5.3.

XI.5.1 Constructing the RejectionABC object

The constructors that can be used to get an instance of the TRejectionABC class are those detailed in Section XI.2.3. As a reminder the prototype available are these ones:

```
// Constructor with a runner
TRejectionABC(TDataServer *tds, TRun *runner, Int_t ns=1, Option_t *option="");
// Constructor with a TCode
TRejectionABC(TDataServer *tds, TCode *code, Int_t ns=1, Option_t *option="");
// Constructor with a function  using Launcher
TRejectionABC(TDataServer *tds, void (*fcn)(Double_t*,Double_t*), const char *varexpinput,  ↩
    const char *varexpoutput, int ns=1, Option_t *option="");;
TRejectionABC(TDataServer *tds, const char *fcn, const char *varexpinput, const char * ↩
    varexpoutput, int ns=1, Option_t *option="");
```

The details about these constructor can be found in Section XI.2.3.1, Section XI.2.3.2 and Section XI.2.3.3 respectively for the TRun, TCode and TLauncherFunction-based constructor. In all cases, the number of samples ns has to set and represents the number of configurations kept in the final sample. An important point is discussed below about the algorithm properties, as to know how many computations will be done, as from our implementation, it actually depends on the percentile value chosen, see Section XI.5.2.1.

As for the option, there is a specific option which might be used to change the default value of the *a posteriori* behaviour. The final sample is a distribution of the parameters value and if one wants to investigate the impact of the *a posteriori* measurement, two possible choice can be made to get a single-point estimate that would best describes the distribution:

- use the mean of the distribution: the **default** option chosen

- use the mode of the distribution: the user needs to add "mode" in the option field of the `TRejectionABC` constructor.

The default solution is straightforward, while the second needs an internal smoothing of the distribution in order to get the best estimate of the mode.

The final step here it to construct the `TDistanceFunction` which is the compulsory step which should always come right after the constructor, but a word of caution about this step:

Warning In the case where you are comparing your reference datasets to a deterministic model (meaning no intrinsic stochastic behaviour is embedded in the code or function) then you might want to specify your uncertainty hypotheses to the method, as discussed below in Section XI.5.2.2.

XI.5.2 Define the TRejectionABC algorithm properties

Once the `TRejectionABC` instance is created along with its `TDistanceFunction`, there are few methods that can be of use in order to tune the algorithm parameters. All these methods are optional in the sens that there are default value, each are detailed in the following sub-sections.

XI.5.2.1 Define the percentile

The first method discussed here is rather simple: the idea behind the rejection is to kept the best configuration tested and this can be done either by looking at the distance results themselves with respect to a threshold value (called δ in [30]) or looking at a certain fraction of configurations, defined through a percentile ε_{Dist}. The latter is the one implemented in the `TRejectionABC` method so fat, the **default** being 1%, which can be written

$$\varepsilon_{Dist} = 0.01.$$

In order to change this, the user might want to call the method

```
void setPercentile(double eps);
```

in which the only argument is the value of the percentile that should be kept.

An important consequence of this is that the number of configurations that will be tested is computed as follow

$$n_{\text{Comp}} = \frac{n_S}{\varepsilon_{Dist}}$$

where n_S is the number of configurations that should be kept at the end.

XI.5.2.2 Introducing noise for deterministic function

As already explained previously, in the case where you are comparing your reference datasets to a deterministic model (meaning no intrinsic stochastic behaviour is embedded in the code or function) then you might want to specify your uncertainty hypotheses to the method. This can be done, by calling the

```
void setGaussianNoise(const char *stdname);
```

The idea here to inject random noise (assumed Gaussian and centred on 0) to the model prediction using internal variable in the reference datasets to set the value of the standard deviation. The only argument is a list of variables that should have the usual shape `"stdvaria1:stdvaria2"` and whose elements represent variable within the reference `TDataServer` whose values are the standard deviation for every single observation points (which can represents experimental uncertainty for instance). This solutions allows three things:

- define a common uncertainty (a general one throughout the observation of the reference datasets) by simply adding an attribute with a `TAttributeFormula` where the formula would be constant.
- use experimental uncertainties are likely to be be provided along the reference values
- Store all hypotheses in the reference `TDataServer` object. For this reason we strongly recommend to save both the parameter and reference datasets at the end of a calibration procedure.

Warning A word of cautious about the string to be passed: the number of variable in the list `stdname` should match the number of output of your code that you are using to calibrate your parameters. Even in the peculiar case where you'll be doing calibration with two outputs, one being free of any kind of uncertainty, then one should add a zero attribute to provide for this peculiar output if the other one needs uncertainty model.

XI.5.3 Look at the results

Finally once the computation is done there are two different methods (apart from looking at the final datasets): the two drawing methods `drawParameters` and `drawResidues` already introduced respectively in Section XI.2.3.6 and Section XI.2.3.7. There are no specific options or visualisation methods to discuss further.

XI.6 The Markov-chain approach

Unlike the Monte-Carlo methods already discussed in Chapter III to obtain design-of-experiments and which usually provides independent samples (which means that the successive observations are statistically independent unless correlation is purposely injected), the Monte-Carlo techniques describe here are called "Markov-chain" and they provide dependent samples as the estimation of the i-Th iteration only depends of the value of the previous one, the (i-1)Th. The method presented here, so far just the Metropolis-Hasting algorithm, is a way to generate *posterior* distributions from a *priori* and data under the assumption that the residues are Gaussian (see [30] for a brief introduction and more useful references).

The way to use our Metropolis-Hasting class is summarised in few key steps here:

1. Get the reference data, the model and its parameters. The parameters to be calibrated must be `TStochasticAttri` inheriting instances. Choose the assessor type you'd like to use and construct the `TMetropHasting` object accordingly with the suitable distance function. Even though this mainly relies on common code, this part is introduced also in Section XI.6.1, in particular for what happened to the distance function (pay attention to the warning bloc)..
2. Provide algorithm properties (not mandatory), to define optional behaviour and precise the uncertainty hypotheses you want, through the methods discussed in Section XI.6.2.
3. Finally the estimation is performed and the results can be extracted or draw with the usual plots. The specificities are discussed in Section XI.6.3.

4. A special step has to be considered for post-processing. As long as you have your TMetropHasting-instance, even though you have exported you parameter TDataServer to be sure that you have it (this should always be done), the you can investigate the quality of the sample through a plot and functions, see Section XI.6.3 for more details.

XI.6.1 Constructing the TMetropHasting object

The constructors that can be used to get an instance of the TMetropHasting class are those detailed in Section XI.2.3. As a reminder the prototype available are these ones:

```
// Constructor with a runner
TMetropHasting(TDataServer *tds, TRun *runner, Int_t ns=1, Option_t *option="");
// Constructor with a TCode
TMetropHasting(TDataServer *tds, TCode *code, Int_t ns=1, Option_t *option="");
// Constructor with a function  using Launcher
TMetropHasting(TDataServer *tds, void (*fcn)(Double_t*,Double_t*), const char *varexpinput, ↩
     const char *varexpoutput, int ns=1, Option_t *option="");;
TMetropHasting(TDataServer *tds, const char *fcn, const char *varexpinput, const char * ↩
    varexpoutput, int ns=1, Option_t *option="");
```

The details about these constructor can be found in Section XI.2.3.1, Section XI.2.3.2 and Section XI.2.3.3 respectively for the TRun, TCode and TLauncherFunction-based constructor. In all cases, the number of samples ns has to set and represents the number of configurations kept in the final sample.

As for the option, there is a specific option which might be used to change the default value of the *a posteriori* behaviour. The final sample is a distribution of the parameters value and if one wants to investigate the impact of the *a posteriori* measurement, two possible choice can be made to get a single-point estimate that would best describes the distribution:

- use the mean of the distribution: the **default** option chosen
- use the mode of the distribution: the user needs to add "mode" in the option field of the TMetropHasting constructor.

The default solution is straightforward, while the second needs an internal smoothing of the distribution in order to get the best estimate of the mode.

The final step here it to construct the TDistanceFunction which is the compulsory step which should always come right after the constructor, but a word of caution about this step:

Warning Whatever the distance function you're choosing, the setDistanceAndReference is locally redefined so that the distance function will only be a TWeightedLSDistanceFunction. As stated previously, the underlying hypotheses here is that the residue under investigation is normally-distributed. The user is requested to provide a guessed uncertainty (which does not have to be constant throughout the reference observation datasets, but could be) to ponderate the difference (as it should reflect the uncertainty coming from the model misknowledge and / or those affecting the observations). This is discussed in details in [30] but the idea is to summarise the ratio of the likelihood (the newly tested configuration, θ_T with respect to the latest one kept θ_k) which allows to get rid of constant factors and should look like this once transformed to its log form:

$$\log \frac{L(\mathbf{y}|\theta_T)}{L(\mathbf{y}|\theta_k)} = \frac{1}{2}\sum_{i=1}^{n}\left(\frac{y_i - f_{\theta_k}(\mathbf{x}_i)}{\sigma_{\varepsilon_i}}\right)^2 - \frac{1}{2}\sum_{i=1}^{n}\left(\frac{y_i - f_{\theta_T}(\mathbf{x}_i)}{\sigma_{\varepsilon_i}}\right)^2.$$

XI.6.2 Define the Metropolis-Hasting algorithm properties

Once the `TMetropHasting` instance is created along with its `TDistanceFunction`, there are few methods that can be of use in order to tune the algorithm parameters. All these methods are optional in the sens that there are default value, each are detailed in the following sub-sections.

XI.6.2.1 Information about the process

The first method discussed here is rather simple: a Markov-Chain method might be rather long to estimate, particularly when running a code, whatever the chosen architecture. This method just change the printing message showing how well the algorithm is converging. The prototype is simply this one

```
void setNbDump(int nbDump);
```

in which the only argument is the modulo to which the algorithm will display where it stands. The **default** value being 1000, the output should starts like this:

```
1000 events done
2000 events done
3000 events done
...
```

XI.6.2.2 Initialising the process

As already explained in [30], the random walk algorithm starts at a given point and then move from there to the new configuration. This means that two things have to be precised for every single parameters: a starting value, and a variation range used to move to the next location. All the input parameters being stochastic (or in other way, they're all instance of `TStochasticDistribution`-inheriting classes), the **default** values are

- randomly drawn for the starting point values;
- the theoretical standard deviation as variation range.

The default behaviour can easily be overcome by calling the `setInitialisation` method with one of these two prototypes:

```
// Protoype with arrays
void setInitialisation(int n, double *values, double *standDev);
// Protoype with vectors
void setInitialisation(vector<double> values, vector<double> standDev);
```

The first one is using simple arrays with a first argument being their size, the second one is safer as the simple of the collection is embedded within the collection itself (the existence of these two prototypes is relevant because of python-binding). Disregarding these technical details, the first argument should contains the starting point of all parameters while the second one contains their variation range meaning that both arrays or vector, should have the size p.

XI.6.2.3 Tune the acceptation rate

As also explained in [30], there are theoretical discussion on the acceptation rate expected, depending on the dimension of the parameter space for instance. As stated in some references (see [22, 23]) when well initialised, a-single dimension problem acceptation rate could be around 44% and this should go down to 23% as the dimension increases. By **default**, the exploration parameters are set by the initialisation step (as discussed in Section XI.6.2.2

If one wants to change this, a possible handle would be to use the `setAcceptationRatioRange` method whose prototype is the following

```
void setAcceptationRatioRange( double lower, double higher);
```

This prototype takes two argument: the lower ($r_{\rm low}$) and higher ($r_{\rm hig}$) bounds. The idea is simply that after a certain number of estimation (called *batch*, whose value is set to 100) the algorithm looks at the acceptation rate achieved (actually this is computed for every configuration and kept in the final `TDataServer` object). If the lower and higher bounds have been set, at the end of a batch, three cases are possibles (when the acceptation rate is called $r_{\rm acc}$):

- $r_{\rm acc} \leq r_{\rm low}$: the acceptation rate is too low with respect to your goal, which means that you might be moving to far when moving from the latest conserved configuration. As a consequence, the variation range is reduced by 10% to help convergence.
- $r_{\rm hig} \leq r_{\rm acc}$: the acceptation rate is too high with respect to your goal, which means that you might not be exploring the parameter space as you should, only investigating a nicely working area. As a consequence, the variation range is increased by 10% to help the exploration.
- $r_{\rm low} \leq r_{\rm acc} \leq r_{\rm hig}$: the acceptance rate is the desired range, so nothing's done.

When the `setAcceptationRatioRange` method has been called, this process will be called at the end of every batch. Otherwise, the **default** behaviour is to let the acceptation rate as it is, has no boundaries have been set.

Few words of caution: obviously, from the prototype and the behaviour discussed above, there are two rules that should be followed when using this method: $(r_{\rm low}, r_{\rm hig}) \in [0,1]^2$ and $r_{\rm low} < r_{\rm hig}$.

Finally, one can have a look at the evolution of the acceptation rate by plotting the information, as discussed in Section XI.6.3.2.

XI.6.3 Look at the results

Finally once the computation is done there are three different kinds of results and several ways to interpret but also transform it. This section details some important and specific aspect to it. The first thing to keep in mind is the fact that this method should lead to a sample of configurations. This sections introduces first quality investigation for the provided sample and then ways to illustrate these results.

Once the algorithm has stopped, the first thing to check is the quality of the sample which can be assessed by different ways. Let's start by introducing the theoretical aspects: when discussing the sample two parameters might be of interest

- the burn-in (also called warm-up): it corresponds to the number of iteration needed to reached a suitable approximation of the target density (see [30]). This has to be defined by the user mainly from the trace plot (discussed below, see Section XI.6.3.1).
- the auto-correlation: the idea behind this is to estimate how self-correlated the samples are. The argument called **lag** can be used to thin the sample, see the discussion in the dedicated section Section XI.6.3.3.

XI.6.3.1 Drawing the trace

The trace plot is simply the evolution of the value of each parameter as a function of the iteration number. The idea is to get a glimpse at two information:

- a possible bad behaviour in the low iteration region that could arise because the algorithm is supposed to converge to the targeted density after a certain number of attempts (see [30]), which should lead to the definition of the **burn-in**
- a possible correlation between the estimations (spacial one) that could give hint to a **lag** estimation if needed but also to increase the burn-in if peculiar behaviour is seen.

The method to do this is called `drawTrace` and as the following prototype:

```
void drawTrace(TString sTitre, const char *variable = "*", const char *select = "1>0",  ↩
    Option_t * option = "");
```

It takes up to four arguments, three of which are optional:

sTitre The title of the plot to be produced (an empty string can be put).

variable This field should contain the parameter list to be drawn (with the usual format of parameter's name splitted by ":"). The default value is to draw all parameters (the `"*"` field).

select A selection field to remove some kept configurations for instance if you want to consider the *burn-in* period or lag procedure.

option The optional field can be used to tune a bit the plots, options being separated by commas

- "nonewcanvas": if this option is used the plot will be done in an existing canvas, if not, a new `TCanvas` will be created on the spot.
- "vertical": if this option is used and more than one parameters have to be displayed, the canvas is splitted into as many windows as parameters in the variable list, the windows being stacked one above the others, using the full width. The default is "horizontal" so the plots are side-by-side.

Taking back the example shown in Section XIV.11.4, one can split the trace plots (originally shown in Figure XIV.102) into two regions using the selection argument. The first guess chosen here is to split events below and above a threshold of 100 events with the following code, leading to Figure XI.2

```
TCanvas *cantr = new TCanvas("cantr","cantr",1200,800);
TPad *pad = new TPad("pad","ppad",0, 0.03, 1, 1);  pad->Draw();  pad->cd();
pad->Divide(1,2);
pad->cd(1);
cal->drawTrace("Looking for burn-in","*",Form("%s < 100",tdsPar->getIteratorName()),"  ↩
    nonewcanvas");
pad->cd(2);
cal->drawTrace("Looking for burn-in","*",Form("%s > 100",tdsPar->getIteratorName()),"  ↩
    nonewcanvas");
```

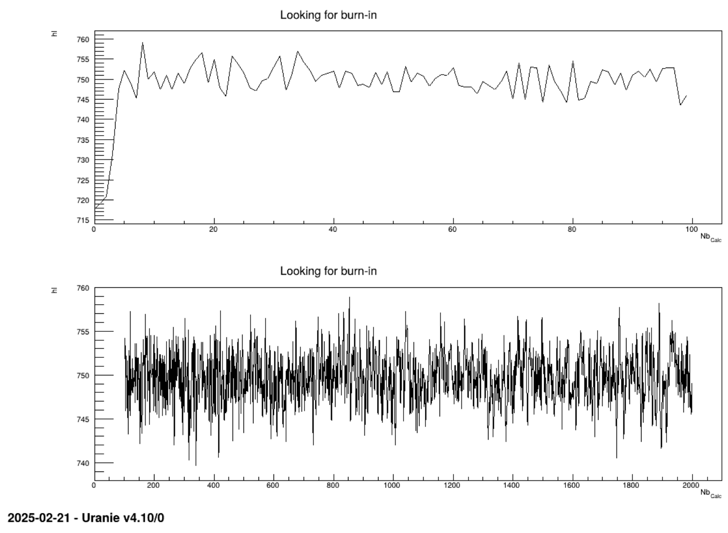


Figure XI.2: Trace distributions split between below and above 100 threshold

From Figure XI.2, one can see, from the top pad, that the very beginning seem wrong, as too far away (probably an initialisation far from the more probable value) and then the behaviour seem fine as it evolves around what's seems to be the most probable value going back and forth. From there a very small burn-in of for example 20 can be chosen. Once this value is chosen it can be provided back to the calibration object and it will be used in many drawing methods as part of the default cut value. This can be done by calling the `setBurnin` method whose signature is the following one

```
void setBurnin(int burnin);
```

Warning Setting a burnin value with the `setBurnin` method will affect most the drawing method, and one should see a line that state what's the default cut used (either burnin or lag selection) to produce a plot. Only the residues plot will not be affected as the *a posteriori* residues are computed during the `estimateParameters` methods, so prior to any burnin or lag determination. If one wants to re-estimate residues with a given set of parameters that takes into account either lag or burnin, use the 3 `estimateCustomResidues` method discussed in Section XI.2.3.5. If one wants to get rid of current cut, use the `clearDefaultCut` method that does not require any argument and whose purpose is to remove both lag and burnin:

```
void clearDefaultCut();
```

XI.6.3.2 Drawing the acceptation ratio

The acceptation ratio plot is simply the evolution of this value for each parameter as a function of the iteration number. The idea is to get a glimpse at two information:

- a possible bad behaviour in the low iteration region that could arise because the algorithm is supposed to converge to the targeted density after a certain number of attempts (see [30]), which might lead to the definition of the **burn-in** (if the acceptance range has not been tuned by calling the method introduced in Section XI.6.2.3).
- check that the acceptance ratio target you'd like to have is achieved (and guess the impact on your variation range guess).

The method to do this is called `drawAcceptationRatio` and as the following prototype:

```
void drawAcceptationRatio(TString sTitre, const char *variable = "*", const char *select = ↩
    "1>0", Option_t * option = "");
```

It takes up to four arguments, three of which are optional:

sTitre The title of the plot to be produced (an empty string can be put).

variable This field should contain the parameter list to be drawn (with the usual format of parameter's name splitted by ":"). The default value is to draw all parameters (the `"*"` field).

select A selection field to remove some kept configurations for instance if you want to consider the *burn-in* period or lag procedure.

option The optional field can be used to tune a bit the plots, options being separated by commas

- "nonewcanvas": if this option is used the plot will be done in an existing canvas, if not, a new `TCanvas` will be created on the spot.
- "vertical": if this option is used and more than one parameters have to be displayed, the canvas is splitted into as many windows as parameters in the variable list, the windows being stacked one above the others, using the full width. The default is "horizontal" so the plots are side-by-side.

This is illustred in Section XIV.11.5 that show another very simple use-case with two paramters to be estimated at once, run in C++ and python.

XI.6.3.3 Estimate the auto-correlation and thin

As stated previously, a correlation between the estimations is expected because of the nature of the sampler: it is a Markov-chain so every new location depends only on its the previous one. Having an uncorrelated sample is a goal if one wants to use the standard deviation of the given sample to extract confidence interval for instance. The usual way to do is to test **lag** value, meaning using 1 events out of **lag** as sample.

To compute this, the method `getAutoCorrelation` whose prototype is

```
void getAutoCorrelation(vector<int> l, vector<double> *out, int cut);
```

It takes thee argument, one being optional:

- a vector of integers containing the lag values one would like to investigate
- a vector of double containing the auto-correlation computed with all the lag values provided. If there are more than one parameters, these auto-correlations are stored for the first parameters, with all lag values, then come the other parameter with their lag value, and so on...

Taking back the example shown in Section XIV.11.4, one can look at the console in Section XIV.11.4.3 to see the values for our simple case which shows and state that a lag of 5 is well enough to get an uncorrelated sample. In this case, the final sample would be thinned to get only one event out of 5, for instance using a selection of this kind

```
int lag=5;
TString thin_cut=Form("(%s %% %d) == 0",tdsPar->getIteratorName(),lag);
```

Warning This approach is sometimes discussed as people might choose very large lag values which is very crude way, because the auto-correlation might arise from a wrong choice of variation range for instance. For more discussion on this, please look at [24]. Once this value is chosen it can be provided back to the calibration object and it will be used in many drawing methods as part of the default cut value. This can be done by calling the `setLag` method whose signature is the following one

```
void setLag(int lag);
```

Warning Setting a lag value with the `setLag` method will affect most the drawing method, and one should see a line that state what's the default cut used (either burnin or lag selection) to produce a plot. Only the residues plot will not be affected as the *a posteriori* residues are computed during the `estimateParameters` methods, so prior to any burnin or lag determination. If one wants to re-estimate residues with a given set of parameters that takes into account either lag or burnin, use the 3 `estimateCustomResidues` method discussed in Section XI.2.3.5. If one wants to get rid of current cut, use the `clearDefaultCut` method that does not require any argument and whose purpose is to remove both lag and burnin:

```
void clearDefaultCut();
```

XI.6.3.4 Drawing the residues

The residues can be drawn with the classical instance of `drawResidues` whose arguments and options have been introduced in details in Section XI.2.3.6, and whose prototype is recalled here for your convenience:

```
void drawResidues(TString sTitre, const char *variable = "*", const char *select = "1>0",  ↩
    Option_t * option = "");
```

As no specific implementations has been done, the only remaining thing to discuss is the way both the *a priori* and *a posteriori* configuration are chosen. The former is just using the initialisation of the parameters meaning that it either comes from the user through the method discussed in Section XI.6.2.2 or, if this method has not been called, it comes from a random drawing following the *a priori* probability density provided. The latter, one the other hand, has also been discussed in the constructor section of this class (see Section XI.6.1). By default, the mean of the posterior will be used, whereas if the option "mode" is precised when constructing the `TMetropHasting` object, the mode would be estimated and used instead.

To get an example of this function, one can have a look at Figure XIV.103.

XI.6.3.5 Drawing the parameters

The parameters can be drawn with the newly-defined instance of `drawParameters` whose prototype is the same as the original one discussed in Section XI.2.3.6.

```
void drawParameters(TString sTitre, const char *variable = "*", const char *select = "1>0", ↩
     Option_t * option = "");
```

The only difference with the usual instance of `drawParameters` defined in `TCalibration` is that the selection field (the third argument of the function) is now always including an extra selection to take into account the burn-in period (to get a discussion on the burn-in definition see Section XI.6.3).

To get an example of this function, one can have a look at Figure XIV.104.

Chapter XII

The Uncertainty modeler module

XII.1 Introduction

We present in this section the quantitative comparison of an already existing sample to a given probability density function ("PDF") in the **libUncertModeler** module. The namespace of this library is **URANIE::UncertModeler**.

XII.2 Tests based on the *Empirical Distribution Function ("EDF tests")*

The implemented tests are the **Kolmogorov-Smirnov** (D) test, the **Cramer-VonMises** (W^2) test and the **Anderson-Darling** (A^2) test. Their aim is to compare a given attribute to a bunch of implemented laws among the following list: the normal, lognormal and uniform law. The details of the computation is given in [30]. The following piece of code gives an example of what can be achieved using these three classes and the usage of their options defined in the summary block below. Figure XII.1 shows the results of such a script.

```
{
 // Create a TDS with 3 kind of distributions
 TDataServer *tds0 = new TDataServer();
 tds0->addAttribute(new TNormalDistribution("n", 1.3, 4.5));
 tds0->addAttribute(new TLogNormalDistribution("ln", 1.3, 4.5));
 tds0->addAttribute(new TUniformDistribution("u", -1.3, 4.5));

 // Create the sample
 TBasicSampling *fsamp = new TBasicSampling(tds0, "lhs", 1000);
 fsamp->generateSample();

 //Create the canvas
 TCanvas *c = new TCanvas("c1","",5,20,1300,600);
 TPad *apad = new TPad("apad","apad",0, 0.03, 1, 1); apad->Draw(); apad->cd();
 apad->Divide(3);

 apad->cd(1);
 TTestKolmogorovSmirnov *tks_n = new TTestKolmogorovSmirnov(tds0, "n");
 TTestCramerVonMises *tcvm_n = new TTestCramerVonMises(tds0, "n");
 TTestAndersonDarling *tad_n = new TTestAndersonDarling(tds0, "n");
 /*Test wrt to a normal distribution whose mu and sigma are taken from
   the original distribution */
 tks_n->computeScore("same:normal");
 tad_n->computeScore("same:normal(0.8,4.5))"); // put wrong mu
 tcvm_n->computeScore("same:normal(1.3,5.5)"); // put wrong sigma
```

```
apad->cd(2);
TTestKolmogorovSmirnov *tks_ln = new TTestKolmogorovSmirnov(tds0, "ln");
TTestCramerVonMises *tcvm_ln = new TTestCramerVonMises(tds0, "ln");
TTestAndersonDarling *tad_ln = new TTestAndersonDarling(tds0, "ln");
/*Test wrt to a lognormal distribution whose mu and sigma are taken from
the original distribution */
tks_ln->computeScore("same:lognormal");
tad_ln->computeScore("same:lognormal(1.2,4.5))"); // put wrong mu
tcvm_ln->computeScore("same:lognormal(1.3,3.5)"); // put wrong sigma

apad->cd(3);
TTestKolmogorovSmirnov *tks_u = new TTestKolmogorovSmirnov(tds0, "u");
TTestCramerVonMises *tcvm_u = new TTestCramerVonMises(tds0, "u");
TTestAndersonDarling *tad_u = new TTestAndersonDarling(tds0, "u");
/*Test wrt to an uniform distribution whose min and max are taken from
  the original distribution (and compared to a normal as well, for fun)*/
tks_u->computeScore("same:uniform");
tad_u->computeScore("same:uniform(-1.2,4.5)"); // put wrong min
tcvm_u->computeScore("same:uniform(-1.3,4.6)"); // put wrong max
 }
```

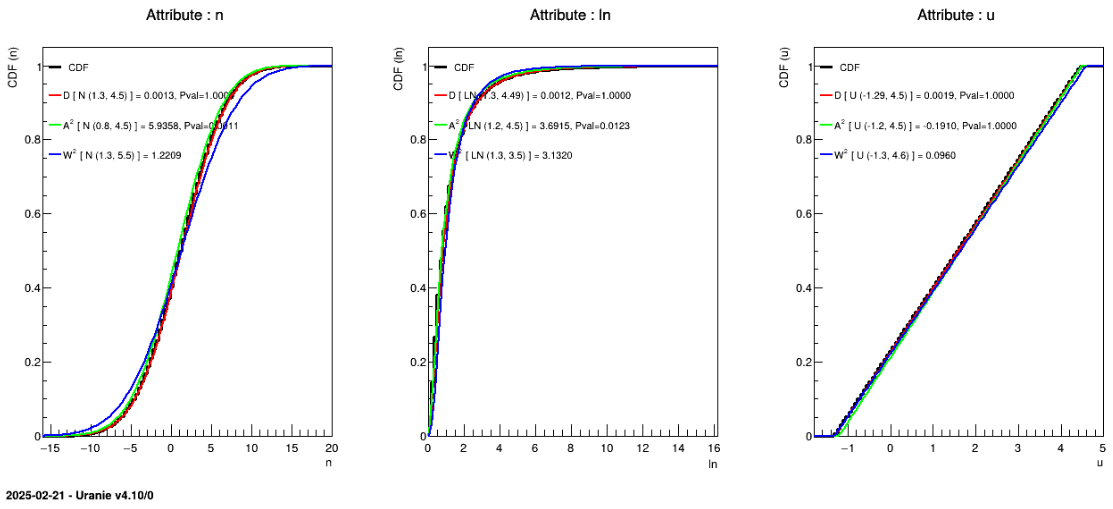


Figure XII.1: Results of the macro defined previously to produce variety of test of already implemented distributions

Summary: Tests based on the EDF (`TTestKolmogorovSmirnov`, `TTestCramerVonMises` and `TTestAndersonDarling` classes)

- `TTestKolmogorovSmirnov`(**`TDataServer`** tds, **const char** * sAtt, **Option_t** * option="")

  Define the **Kolmogorov-Smirnov (D)** test for the attribute *sAtt*. No option is used.

- `TTestCramerVonMises`(**`TDataServer`** tds, **const char** * sAtt, **Option_t** * option="")

  Define the **Cramer-VonMises (W2)** test for the attribute *sAtt*. No option is used.

- `TTestAndersonDarling`(**`TDataServer`** tds, **const char** * sAtt, **Option_t** * option="")

  Define the **Anderson-Darling (A2)** test for the attribute *sAtt*. No option is used.

- `computeScore`(**Option_t*** option="")

  Compute the score for the current test (D, W2 and A2) with laws defined in the option parameter. The laws are separated by the ":" character. If the parameters of the law are defined by the user, they are defined in brackets *"()"* as *normal(30576,1450)*. If the parameters must be estimated by the algorithm, do not use the brackets as *normal*. Actually, the laws implemented are the *normal* and *lognormal* laws.

XII.3 The *Circe* method

The Circe method is a statistical approach which is applied as an alternative to the expert judgement, used to determine the uncertainty of physical model's parameters. These uncertainties can be tricky to estimate as some of these parameters might not be directly measurable. However, it might be possible to use SET (separate-effect tests) experiments, which are sensitive to the physical model, to derive an estimation of these uncertainties. This method has been implemented in Uranie throughout the `TCirce` class. A description of the method is provided in [30]

Summary: `TCirce` class

- `TCirce`(**`TDataServer`** tds, **const char** * Ystar, **const char** * Yhat, **const char** * Derivatives, **const char** * YstarUncert="", **Option_t** * option="")

  Define the *Circe* method on the `TDataServer tds` where the experimental attribute is *Yhat*, the output of the code is *Ystar* and the derivatives from each parameter are specified in the *Derivatives* (list of attributes separated by the "," character).

- `estimate`(**Option_t*** option="")

  Launch the process to estimate the vector of bias and the correlation matrix. No option is used.

- `setTolerance`(**Double_t** dtol)

  Set the tolerance parameter to stop the algorithm. The default value is 1.0e-5.

- `setBVectorInitial`(**`TVectorD`** vec)

  Set the initial B Vector which default vector is the null vector.

- `setCMatrixInitial`(**`TMatrixD`** mat)

  Set the initial C matrix which default matrix is the identity matrix.

- `setNCMatrix`(**Int_t** n)

  Set the number of loops to execute the algorithm with different initials C matrix. The first one is always the identity matrix and the default value is one.

- Double_t `getLikelihood`()

  Get the likelihood for the retained iteration.

- TVectorD `getBVector`()

  Get the B vector for the retained iteration.

- TMatrixD `getCMatrix`()

  Get the C matrix for the retained iteration.

Chapter XIII

The Reliability module

XIII.1 Introduction

So far, the Reliability module only provides basic classes for FORM-SORM studies using the Relauncher architecture.

XIII.2 Form Sorm

The FORM (First Order Reliability Method) and SORM (Second Order Reliability Method) try to estimate the probability of failure of a system. It is often used when a monte-carlo approach is not affordable due to evaluation cost.

In short, its principle uses the notion of *design point*: the most probable point that exceeds safety threshold. A probability transformation is applied to pass from physical space to a gaussian space, where all random variables are standard normal ones, statistically independent one from another. In this space, FORM approximates the separate function between safe and unsafe solutions with a tangent hyper-plane, while SORM approximates it with a second order taylor expansion. From this design point, FORM estimation is directly calculable, while SORM needs to evaluate the curves around it. Design point search is treated as a constrained optimisation problem.

Currently, Multi-FORM and Multi-SORM are not available, as well as the conjoint use of *importance sampling* with FORM SORM.

A FORM-SORM problem is a kind of parametric studies. So, it could make the best of the Relauncher architecture. Having this in mind, having a look at Chapter VIII is crucial in order to get good understanding of the following, already-introduced, concepts.

XIII.2.1 Study outline

Before getting into the heart of its study, the user has to define the reliability function, which determines if the system is secure or not. The standard steps to make a FORM SORM studies are:

1. define the problem
   - declare the input parameters with their statistical law.
   - choose the probability transformation between random spaces.
   - define the `TEval` providing the reliability function.
   - compose the two previous functions in a `TSormEval`.

2. find the design point (optimisation problem)
   - define a `TDataServer`
   - choose an optimisation solver (eventually configure it),
   - create the corresponding master (eventually configure it), and declare the objective and constraints of the problem.
   - run the optimisation.
3. (FORM estimation is directly accessible from the last step in the `TDataServer`)
4. calculate SORM estimation
   - define the `TSorm` object.
   - run the estimation.
5. and finally, analyse the results.

To match the relauncher mold, separating evaluation from study, all parts but first are on the study side.

The second part deals with a constraint optimisation problem: find the most probable solution (objective) that is unsafe (constraint). As the transformation function does not provide the gradient matrix and problem is constrained, the `TNloptCobyla` algorithm is the recommended local solver. See the Reoptimizer chapter for details.

The FORM estimation is implicit and the SORM estimation is optional. The classes dedicated to FORM SORM studies are described in the next chapter.

XIII.2.2 TSimpleTransform

Currently, the only available probability transformation in Uranie is the simplest one. It supposes that all random variables are independent and have its statistical law known.

`TSimpleTransform` is a `TEval` subclass: it does not returned the gradient matrix. Its constructor has no argument. The `addParameter` method adds the physical random variables.

In fact, it is the inverse transformation that is implemented. The inputs are the normal variable values, and it computes the physical values, the Hasofer-Lind indicator, and the FORM estimation.

XIII.2.3 TFormEval

TFormEval is a `TComposeEval` subclass used to compose the probability transformation and the safety function. The constructor have two arguments corresponding to this two functions. To complete the definition, the `addConstraint` method select the output variable used to classify safe and unsafe items. It looks like the `addConstraint` method of Reoptimizer classes and has the same arguments: a `TAttribute` and an optional modifier.

This class has a helper method (`addOjective`) to declare both the objective and the constraint to the design point optimisation problem.

XIII.2.4 TSorm

The SORM estimation needs the principal curves of the design point. The implemented method approximates these curve: for each half axe of the hyper-plane, it searches the border line points of the separation function. The returned probability uses the Breitung approximation.

The `TSorm` class is a `TMaster` subclass. The constructor has the two standard arguments: a `TDataServer` and a `TRun`.

For each design points found in the `TDataServer`, the `solverLoop` method performs an estimation of the curves, and completes its data with the FORM correction factor and SORM estimation. If they are many design points in the `TDataServer`, it may be useful to filter them to perform a SORM estimation only with principal ones.

These searches can be done in parallel using an adequate `TRun`. It does not exploit if there is many points (it is often not the case), but inside a point estimation, can treat each half axe searches in parallel. For a 6 input variable problem, 10 evaluation resources may be exploited.

Chapter XIV

Use-cases in C++

XIV.1 Introduction

Several use-cases of Uranie are described in this chapter with a small description of the implemented methods in the Uranie platform. These macros are located in the sub-directory *"/share/uranie/macros"* of the installation folder of Uranie (*$URANIESYS*).

In this chapter, inside each macro, the Uranie specific **namespaces** might not always be specified anymore. These specifications are gathered in the `rootlogon.C` file, introduced in Section I.2.4, which is automatically loaded when executing **root**. An example would be:

```
using namespace URANIE::DataServer;
using namespace URANIE::Launcher;
using namespace URANIE::Sampler;
using namespace URANIE::Optimizer;
using namespace URANIE::Modeler;
using namespace URANIE::UncertModeler;
using namespace URANIE::Sensitivity;
using namespace URANIE::Relauncher;
using namespace URANIE::Reoptimizer;
using namespace URANIE::Calibration;
using namespace URANIE::Reliability;
// using namespace URANIE::XMLProblem;
// using namespace URANIE::MpiRelauncher;

void rootlogon()
{

    gStyle->SetPalette(1);
    gStyle->SetOptDate(21);

    //General graphical style
    // Default colors
    int white = 0;
    int color = 30;

    //Legend
    gStyle->SetLegendBorderSize(0);
    gStyle->SetFillStyle(0);

    // Pads
    gStyle->SetPadColor(white);
    gStyle->SetTitleFillColor(white);
    gStyle->SetStatColor(white);

}

/* ==================== Hint ====================

   Might be practical to store this in a convenient place (for instance
   your home directory) and to create an alias to make sure that you use
   only one rootlogon file independently of where you are.

   example : alias root="root -l ${HOME}/rootlogon.C"

   Many style issue can be set once and for all here.

   Warnings :
    => The name of the main function (in between the void and the () part)
    has to be the same as the name of the file (without extension).
    => If you intend to change this file name and make it a hidden file (let's
    say  ${HOME}/.toto.C, the name of the main function would have to start with
    an underscore, so here it would be "void _toto()".
*/
```

XIV.2 Macros DataServer

In a first step, to get accustomed with `TDataServer` , we propose different macros related to this subject. Since it constitutes the preliminary and almost mandatory step of a proper use of Uranie, these macros are only for educational purposes.

XIV.2.1 Macro "`dataserverAttributes.C`"

XIV.2.1.1 Objective

The goal of this macro is only to master the objects `TAttribute` and `TDataServer` of Uranie. Three attributes will be created and linked to a `TDataServer` object, then the *log* of this object will be printed to check internal data of this `TDataServer`.

XIV.2.1.2 Macro Uranie

```
{

  // Define the attribute "x"
  TAttribute *px = new TAttribute("x", -2.0, 4.0);
  px->setTitle("#Delta P^{#sigma}");
  px->setUnity("#frac{mm^{2}}{s}");

  // Define the attribute "y"
  TAttribute *py = new TAttribute("y", 0.0, 1.0);

  // Define the DataServer of the study
  TDataServer *tds = new TDataServer("tds", "my first TDS");
  // Add the attributes in the TDataServer
  tds->addAttribute(px);
  tds->addAttribute(py);
  tds->addAttribute(new TAttribute("z", 0.25, 0.50));

  tds->printLog();

}
```

The first attribute *"x"* is defined on [-2.0, 4.0]; its title is ΔP^{σ} and unity $\frac{mm^2}{s}$

```
TAttribute *px = new TAttribute("x", -2.0, 4.0);
px->setTitle("#Delta P^{#sigma}");
px->setUnity("#frac{mm^{2}}{s}");
```

The second attribute *"y"* is defined on [0.0,1.0]; it will be set with its name as title but without unity.

```
TAttribute *py = new TAttribute("y", 0.0, 1.0);
```

Secondly, a `TDataServer` object is created and the two attributes *x* and *y* created before are linked to this one.

```
TDataServer *tds = new TDataServer("tds", "my first TDS");
tds->addAttribute(px);
tds->addAttribute(py);
```

Finally, the last attribute *z* (defined on [0.25,0.50]) is directly added to the `TDataServer` (its title will be its name and it will be set without unity) by creating it. An attribute could, indeed, be added to a `TDataServer` meanwhile creating it, but then no other information than those available in the constructor would be set.

```
tds->addAttribute(new TAttribute("z", 0.25, 0.50));
```

Then, the *log* of the `TDataServer` object is printed.

```
tds->printLog();
```

Generally speaking, all Uranie objects have the `printLog` method which allows to print internal data of the object.

XIV.2.1.3 Console

```
Processing dataserverAttributes.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

TDataServer::printLog[]
Name[tds] Title[my first TDS]
Origin[Unknown]
 _sdatafile[]
 _sarchivefile[_dataserver_.root]
*******************************
** TDataSpecification::printLog  ******
**   Name[uheader__tds] Title[Header of my first TDS]
**   relationName[Header of my first TDS]
**   attributs[4]
**** With _listOfAttributes
 Attribute[0/4]
*******************************
** TAttribute::printLog  ******
**    Name[tds__n__iter__]
**   Title[tds__n__iter__]
**   unity[]
**    type[0]
**   share[1]
**   Origin[kIterator]
**   Attribute[kInput]
**   _snote[]
--------------------------------------------------------------------------------------------- ↩

--------------------------------------------------------------------------------------------- ↩

** - min[] max[]
** - mean[] std[]
**   lowerBound[-1.42387e+64] upperBound[1.42387e+64]
** NOT _defaultValue[]
** NOT _stepValue[]
 ** No Attribute to substitute level[0]
*******************************
 Attribute[1/4]
*******************************
** TAttribute::printLog  ******
**    Name[x]
**   Title[#Delta P^{#sigma}]
**   unity[#frac{mm^{2}}{s}]
**    type[0]
**   share[2]
**   Origin[kAttribute]
**   Attribute[kInput]
**   _snote[]
--------------------------------------------------------------------------------------------- ↩

--------------------------------------------------------------------------------------------- ↩

** - min[] max[]
** - mean[] std[]
**   lowerBound[-2] upperBound[4]
** NOT _defaultValue[]
** NOT _stepValue[]
 ** No Attribute to substitute level[0]
*******************************
 Attribute[2/4]
*******************************
** TAttribute::printLog  ******
**    Name[y]
**   Title[y]
**   unity[]
**    type[0]
**   share[2]
**   Origin[kAttribute]
**   Attribute[kInput]
**   _snote[]
--------------------------------------------------------------------------------------------- ↩

--------------------------------------------------------------------------------------------- ↩

** - min[] max[]
** - mean[] std[]
**   lowerBound[0] upperBound[1]
** NOT _defaultValue[]
** NOT _stepValue[]
 ** No Attribute to substitute level[0]
*******************************
```

```
 Attribute[3/4]
*******************************
** TAttribute::printLog  ******
**    Name[z]
**   Title[z]
**   unity[]
**    type[0]
**   share[2]
**   Origin[kAttribute]
**   Attribute[kInput]
**   _snote[]
--------------------------------------------------------------------------------------------- ↩

--------------------------------------------------------------------------------------------- ↩

** - min[] max[]
** - mean[] std[]
**   lowerBound[0.25] upperBound[0.5]
** NOT _defaultValue[]
** NOT _stepValue[]
 ** No Attribute to substitute level[0]
*******************************
** TDataSpecification::fin de printLog *******************************
fin de TDataServer::printLog[]
```

XIV.2.2 Macro "dataserverMerge.C"

XIV.2.2.1 Objective

The objective of this macro is to merge data contained in a `TDataServer` with data contained in another `TDataServer`. Both `TDataServer` have to contain the same number of patterns. We choose here to merge two `TDataServer`, loaded from two ASCII files, each of which contains 9 patterns.

The first ASCII file "`tds1.dat`" defines the four variables **x, dy, z, theta**:

```
#COLUMN_NAMES: x| dy| z| theta
#COLUMN_TITLES: x_{n}| "#delta y"| ""| #theta
#COLUMN_UNITS: N| Sec| KM/Sec| M^{2}

1 1 11 11
1 2 12 21
1 3 13 31
2 1 21 12
2 2 22 22
2 3 23 32
3 1 31 13
3 2 32 23
3 3 33 33
```

and the second ASCII file "`tds2.dat`" defines the four other variables **x2, y, u, ua**:

```
#COLUMN_NAMES: x2| y| u| ua

1 1 102 11
1 2 104 12
1 3 106 13
2 1 202 21
2 2 204 22
2 3 206 23
```

```
3 1 302 31
3 2 304 32
3 3 306 33
```

The merging operation will be executed in the first `TDataServer` `tds1`; so it will contain all the attributes at the end.

XIV.2.2.2 Macro Uranie

```
{

  TDataServer * tds1 = new TDataServer();
  TDataServer * tds2 = new TDataServer();
  tds1->fileDataRead("tds1.dat");
  cout<<"Dumping tds1"<<endl;
  tds1->Scan("*");

  tds2->fileDataRead("tds2.dat");
  cout<<"Dumping tds2"<<endl;
  tds2->Scan("*");

  tds1->merge(tds2);
  cout<<"Dumping merged tds1 and tds2"<<endl;
  tds1->Scan("*","","colsize=3 col=9::::::::");

}
```

Both TDataServers are filled with ASCII data files with the method `filedataRead()`.

```
tds1->fileDataRead("tds1.dat");
tds2->fileDataRead("tds2.dat");
```

Data of the second dataserver `tds2` are then merged into the first one.

```
tds1->merge(tds2);
```

Data are then dumped in the terminal:

```
tds1->Scan("*");
```

XIV.2.2.3 Console

```
Processing dataserverMerge.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

Dumping tds1
************************************************************************
*    Row   * tds1__n__ *       x.x *     dy.dy *       z.z * theta.the *
************************************************************************
*        0 *         1 *         1 *         1 *        11 *        11 *
*        1 *         2 *         1 *         2 *        12 *        21 *
*        2 *         3 *         1 *         3 *        13 *        31 *
*        3 *         4 *         2 *         1 *        21 *        12 *
*        4 *         5 *         2 *         2 *        22 *        22 *
*        5 *         6 *         2 *         3 *        23 *        32 *
*        6 *         7 *         3 *         1 *        31 *        13 *
*        7 *         8 *         3 *         2 *        32 *        23 *
*        8 *         9 *         3 *         3 *        33 *        33 *
************************************************************************
Dumping tds2
************************************************************************
*    Row   * tds2__n__ *     x2.x2 *       y.y *       u.u *     ua.ua *
************************************************************************
*        0 *         1 *         1 *         1 *       102 *        11 *
*        1 *         2 *         1 *         2 *       104 *        12 *
*        2 *         3 *         1 *         3 *       106 *        13 *
*        3 *         4 *         2 *         1 *       202 *        21 *
*        4 *         5 *         2 *         2 *       204 *        22 *
*        5 *         6 *         2 *         3 *       206 *        23 *
*        6 *         7 *         3 *         1 *       302 *        31 *
*        7 *         8 *         3 *         2 *       304 *        32 *
*        8 *         9 *         3 *         3 *       306 *        33 *
************************************************************************
Dumping merged tds1 and tds2
************************************************************************
*    Row   * tds1__n__ * x.x * dy. * z.z * the * x2. * y.y * u.u * ua. *
************************************************************************
*        0 *         1 *   1 *   1 *  11 *  11 *   1 *   1 * 102 *  11 *
*        1 *         2 *   1 *   2 *  12 *  21 *   1 *   2 * 104 *  12 *
*        2 *         3 *   1 *   3 *  13 *  31 *   1 *   3 * 106 *  13 *
*        3 *         4 *   2 *   1 *  21 *  12 *   2 *   1 * 202 *  21 *
*        4 *         5 *   2 *   2 *  22 *  22 *   2 *   2 * 204 *  22 *
*        5 *         6 *   2 *   3 *  23 *  32 *   2 *   3 * 206 *  23 *
*        6 *         7 *   3 *   1 *  31 *  13 *   3 *   1 * 302 *  31 *
*        7 *         8 *   3 *   2 *  32 *  23 *   3 *   2 * 304 *  32 *
*        8 *         9 *   3 *   3 *  33 *  33 *   3 *   3 * 306 *  33 *
************************************************************************
```

XIV.2.3 Macro "dataserverLoadASCIIFilePasture.C"

XIV.2.3.1 Objective

The objective of this macro is to load two `TDataServer` objects using two different ways: either with an ASCII file `"pasture.dat"` or with a design-of-experiments. Then, we evaluate the analytic function `ModelPasture` on this two `TDataServer`. The data file `"pasture.dat"` is written in the "Salome-table" format of Uranie:

```
#COLUMN_NAMES: time| yield

9 8.93
14 10.8
21 18.59
28 22.33
42 39.35
57 56.11
63 61.73
70 64.62
79 67.08
```

XIV.2.3.2 Macro Uranie

```
#include "TMath.h"

void ModelPasture(Double_t *x, Double_t *y)
{
  Double_t theta1=69.95, theta2=61.68, theta3=-9.209, theta4=2.378;

  y[0] = theta1;
  y[0] -= theta2* TMath::Exp( -1.0 * TMath::Exp( theta3 + theta4 * TMath::Log(x[0])));
}

void dataserverLoadASCIIFilePasture()
{

  TCanvas *C = new TCanvas("mycanvas","mycanvas",1);

  TDataServer* tds = new TDataServer();
  tds->fileDataRead("pasture.dat");

  tds->getTuple()->SetMarkerStyle(8);
  tds->getTuple()->SetMarkerSize(1.5);
  tds->draw("yield:time");

  TLauncherFunction *tlf = new TLauncherFunction(tds, ModelPasture,"time","yhat");
  tlf->run();

  tds->getTuple()->SetMarkerColor(kBlue);
  tds->getTuple()->SetLineColor(kBlue);

  tds->draw("yhat:time","","lpsame");

  TDataServer *tds2 = new TDataServer();
  tds2->addAttribute( new TUniformDistribution("time2",9, 80));

  TSampling *tsamp = new TSampling(tds2, "lhs", 1000);
  tsamp->generateSample();

  tds2->getTuple()->SetMarkerColor(kGreen);
  tds2->getTuple()->SetLineColor(kGreen);
  tlf = new TLauncherFunction(tds2, ModelPasture,"","yhat2");
  tlf->run();


  tds2->draw("yhat2:time2","","psame");
  tds->draw("yhat:time","","lpsame");

  gPad->SaveAs("pasture.png");

}
```

The design `ModelPasture` is defined in a function

```
void ModelPasture(Double_t *x, Double_t *y)
{
  Double_t theta1=69.95, theta2=61.68, theta3=-9.209, theta4=2.378;

  y[0] = theta1;
  y[0] -= theta2* TMath::Exp( -1.0 * TMath::Exp( theta3 + theta4 * TMath::Log(x[0])));
}
```

The first `TDataServer` is filled with the ASCII file "`pasture.dat`" through the `fileDataRead` method

```
tds->fileDataRead("pasture.dat");
```

The design is evaluated with the function `ModelPasture` applied on the input attribute *time*, leading to the output attribute named *yhat*.

```
TLauncherFunction *tlf = new TLauncherFunction(tds, ModelPasture,"time","yhat");
tlf->run();
```

A `TAttribute`, obeying an uniform law on [9;80] is added to the second `TDataServer` which is filled with a design-of-experiments of 1000 patterns, using the LHS method.

```
tds2->addAttribute( new TUniformDistribution("time2",9, 80));
TSampling *tsamp = new TSampling(tds2, "lhs", 1000);
tsamp->generateSample();
```

The design is now evaluated with this `TDataServer` on the attribute **time2**

```
tlf = new TLauncherFunction(tds2, ModelPasture,"","yhat2");
tlf->run();
```

XIV.2.3.3 Graph

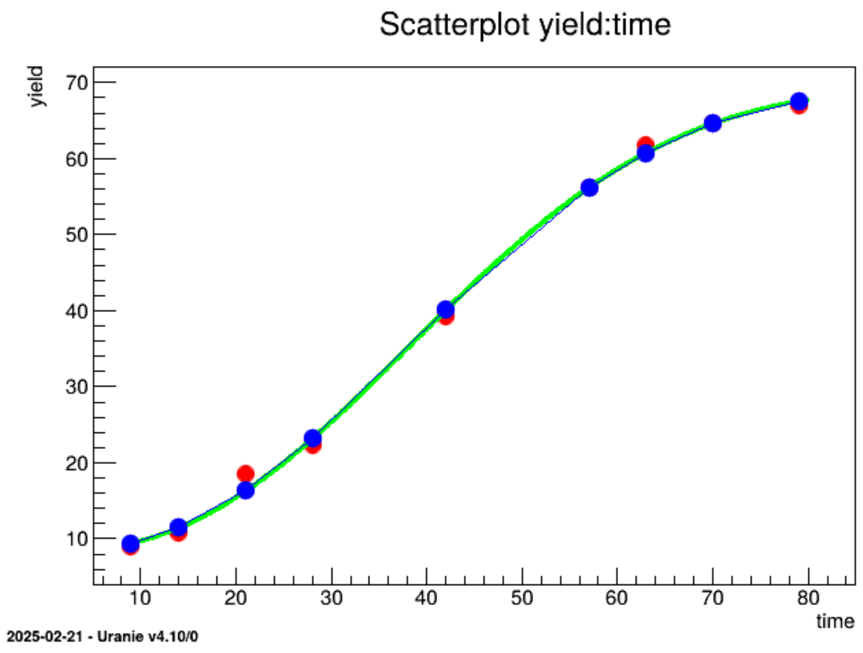


Figure XIV.1: Graph of the macro "`dataserverLoadASCIIFilePasture.C`"

XIV.2.3.4 Console

```
Processing loadASCIIFilePasture.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

Info in <TCanvas::Print>: png file pasture.png has been created
```

XIV.2.4 Macro "dataserverLoadASCIIFile.C"

XIV.2.4.1 Objective

Loading data in a `TDataServer`, using the "Salome-table" format of Uranie and applying basic visualisation methods on attributes.

The data file is named "`flowrateUniformDesign.dat`" and data correspond to an *Uniform* design-of-experiments of 32 patterns for a "code" with 8 inputs (r_ω, r, T_u, T_l, H_u, H_l, L, K_ω) along with a response (*"yhat"*). The data file "`flowrateUniformDesign.dat`" is in the "Salome-table" format of Uranie.

```
#NAME: flowrateborehole
#TITLE: Uniform design of flow rate borehole problem proposed by Ho and Xu(2000)
#COLUMN_NAMES: rw| r| tu| tl| hu| hl| l| kw | ystar
#COLUMN_TITLES: r_{#omega}| r | T_{u} | T_{l} | H_{u} | H_{l} | L | K_{#omega} | y^{*}
#COLUMN_UNITS: m | m | m^{2}/yr | m^{2}/yr | m | m | m | m/yr | m^{3}/yr

0.0500 33366.67  63070.0 116.00 1110.00 768.57 1200.0 11732.14  26.18
0.0500   100.00  80580.0  80.73 1092.86 802.86 1600.0 10167.86  14.46
0.0567   100.00  98090.0  80.73 1058.57 717.14 1680.0 11106.43  22.75
0.0567 33366.67  98090.0  98.37 1110.00 734.29 1280.0 10480.71  30.98
0.0633   100.00 115600.0  80.73 1075.71 751.43 1600.0 11106.43  28.33
0.0633 16733.33  80580.0  80.73 1058.57 785.71 1680.0 12045.00  24.60
0.0700 33366.67  63070.0  98.37 1092.86 768.57 1200.0 11732.14  48.65
0.0700 16733.33 115600.0 116.00  990.00 700.00 1360.0 10793.57  35.36
0.0767   100.0  115600.0  80.73 1075.71 751.43 1520.0 10793.57  42.44
0.0767 16733.33  80580.0  80.73 1075.71 802.86 1120.0  9855.00  44.16
0.0833 50000.00  98090.0  63.10 1041.43 717.14 1600.0 10793.57  47.49
0.0833 50000.00 115600.0  63.10 1007.14 768.57 1440.0 11419.29  41.04
0.0900 16733.33  63070.0 116.00 1075.71 751.43 1120.0 11419.29  83.77
0.0900 33366.67 115600.0 116.00 1007.14 717.14 1360.0 11106.43  60.05
0.0967 50000.00  80580.0  63.10 1024.29 820.00 1360.0  9855.00  43.15
0.0967 16733.33  80580.0  98.37 1058.57 700.00 1120.0 10480.71  97.98
0.1033 50000.00  80580.0  63.10 1024.29 700.00 1520.0 10480.71  74.44
0.1033 16733.33  80580.0  98.37 1058.57 820.00 1120.0 10167.86  72.23
0.1100 50000.00  98090.0  63.10 1024.29 717.14 1520.0 10793.57  82.18
0.1100   100.00  63070.0  98.37 1041.43 802.86 1600.0 12045.00  68.06
0.1167 33366.67  63070.0 116.00  990.00 785.71 1280.0 12045.00  81.63
0.1167   100.00  98090.0  98.37 1092.86 802.86 1680.0  9855.00  72.5
0.1233 16733.33 115600.0  80.73 1092.86 734.29 1200.0 11419.29 161.35
0.1233 16733.33  63070.0  63.10 1041.43 785.71 1680.0 12045.00  86.73
0.1300 33366.67  80580.0 116.00 1110.00 768.57 1280.0 11732.14 164.78
0.1300   100.00  98090.0  98.37 1110.00 820.00 1280.0 10167.86 121.76
0.1367 50000.00  98090.0  63.10 1007.14 820.00 1440.0 10167.86  76.51
0.1367 33366.67  98090.0 116.00 1024.29 700.00 1200.0 10480.71 164.75
0.1433 50000.00  63070.0 116.00  990.00 785.71 1440.0  9855.00  89.54
0.1433 50000.00 115600.0  63.10 1007.14 734.29 1440.0 11732.14 141.09
0.1500 33366.67  63070.0  98.37  990.00 751.43 1360.0 11419.29 139.94
0.1500   100.00 115600.0  80.73 1041.43 734.29 1520.0 11106.43 157.59
```

XIV.2.4.2 Macro Uranie

```
{
  // Create a TDataServer
  TDataServer * tds = new TDataServer();
  // Load the data base in the DataServer
  tds->fileDataRead("flowrateUniformDesign.dat");

  // Graph
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro loadASCIIFile",5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2,2);

  pad->cd(1); tds->draw("ystar");
  pad->cd(2); tds->draw("ystar:rw");
  pad->cd(3); tds->drawTufte("ystar:rw");
  pad->cd(4); tds->drawProfile("ystar:rw");

  tds->startViewer();

}
```

XIV.2.4.3 Graph

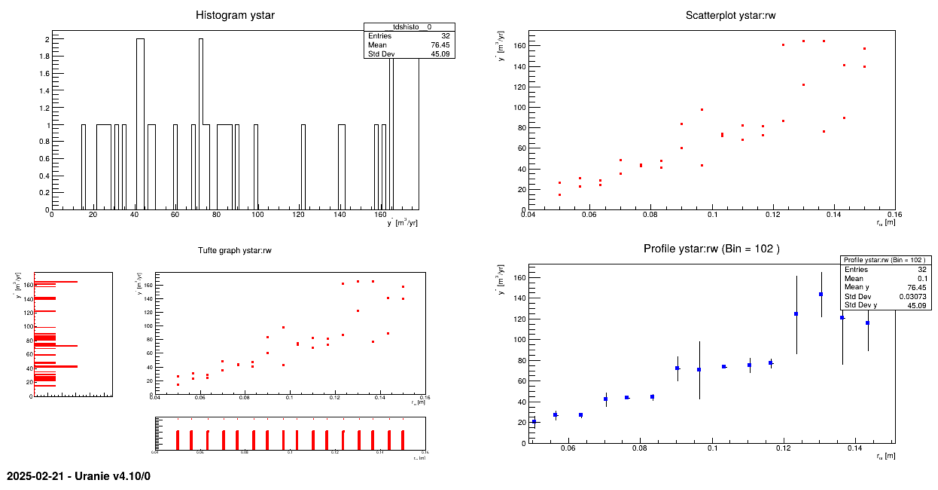


Figure XIV.2: Graph of the macro "dataserverLoadASCIIFile.C"

XIV.2.5 Macro "dataserverLoadASCIIFileYoungsModulus.C"

XIV.2.5.1 Objective

The objective of this macro is to load the ASCII data file "youngsmodulus.dat" and to apply visualisations on the attribute **E** with different options. The data file "youngsmodulus.dat" is in the "Salome-table" format of Uranie.

```
#NAME: youngsmodulus
#TITLE: Young’s Modulus E for the Golden Gate Bridge
#COLUMN_NAMES: E
#COLUMN_TITLES: Young’s Modulues
#COLUMN_UNITS: ksi

28900
```

```
29200
27400
28700
28400
29900
30200
29500
29600
28400
28300
29300
29300
28100
30200
30200
30300
31200
28800
27600
29600
25900
32000
33400
30600
32700
31300
30500
31300
29000
29400
28300
30500
31100
29300
27400
29300
29300
31300
27500
29400
```

Data are then exported in header file "`youngsmodulus.h`" which can be imported in some C file:

```
// File "youngsmodulus.h" generated by ROOT v5.34/32
// DateTime Tue Nov  3 10:40:13 2015
// DataServer : Name="youngsmodulus" Title="Young's Modulus E for the Golden Gate Bridge"  ↩
    Global Select=""

#define youngsmodulus_nPattern 41

//  Attribute Name="E"
Double_t E[youngsmodulus_nPattern] = {
 2.890000000e+04,
 2.920000000e+04,
 2.740000000e+04,
 2.870000000e+04,
 2.840000000e+04,
 2.990000000e+04,
 3.020000000e+04,
 2.950000000e+04,
 2.960000000e+04,
 2.840000000e+04,
 2.830000000e+04,
 2.930000000e+04,
 2.930000000e+04,
 2.810000000e+04,
 3.020000000e+04,
 3.020000000e+04,
 3.030000000e+04,
 3.120000000e+04,
 2.880000000e+04,
 2.760000000e+04,
 2.960000000e+04,
 2.590000000e+04,
 3.200000000e+04,
 3.340000000e+04,
 3.060000000e+04,
 3.270000000e+04,
 3.130000000e+04,
 3.050000000e+04,
 3.130000000e+04,
 2.900000000e+04,
 2.940000000e+04,
 2.830000000e+04,
 3.050000000e+04,
 3.110000000e+04,
 2.930000000e+04,
 2.740000000e+04,
 2.930000000e+04,
 2.930000000e+04,
 3.130000000e+04,
 2.750000000e+04,
 2.940000000e+04,
};
// End of attribute E

// End of File youngsmodulus.h
```

XIV.2.5.2 Macro Uranie

```
{

  TDataServer * tds = new TDataServer();
  tds->fileDataRead("youngsmodulus.dat");
  //  gEnv->SetValue("Hist.Binning.1D.x", 10);
  //   tds->getTuple()->Draw("E>>Attribute E(6, 25000, 34000)","","text");
  //   tds->getTuple()->Draw("E>>Attribute E(16, 25000, 34000)");

  tds->computeStatistic("E");
  tds->getAttribute("E")->printLog();

  tds->exportDataHeader("youngsmodulus.h");

  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro loadASCIIFile",5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2,2);

  pad->cd(1); tds->draw("E");
  pad->cd(2); tds->draw("E" ,"", "nclass=sturges");
  pad->cd(3); tds->draw("E" ,"", "nclass=scott");
  pad->cd(4); tds->draw("E" ,"", "nclass=fd");
}
```

The `TDataServer` is filled with the ASCII data file `"youngsmodulus.dat"` with the method `fileDataRead`:

```
tds->fileDataRead("youngsmodulus.dat");
```

Variable **E** is then visualised with different options:

```
tds->draw("E" ,"", "nclass=sturges");
tds->draw("E" ,"", "nclass=scott");
tds->draw("E" ,"", "nclass=fd");
```

Characteristic values are computed (maximum, minimum, mean and standard deviation) with:

```
tds->computeStatistic("E");
```

Data are exported in a header file with

```
tds->exportDataHeader("youngsmodulus.h");
```

XIV.2.5.3 Graph

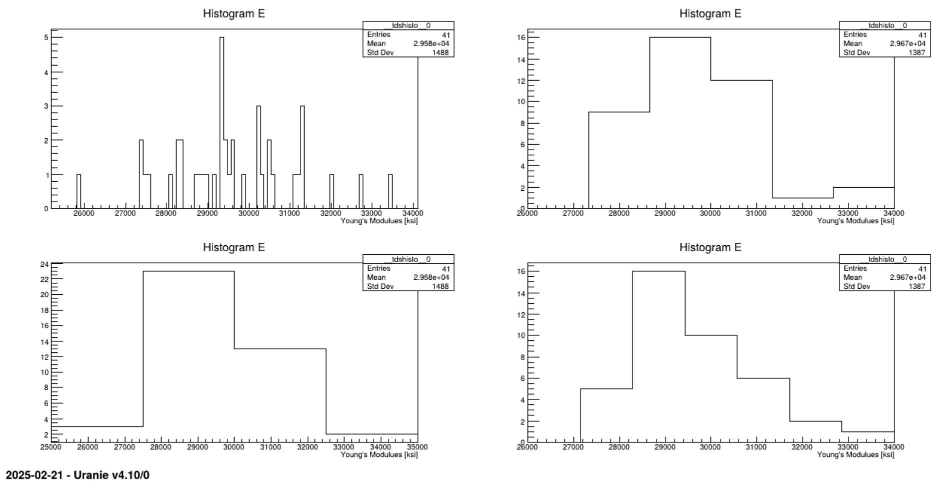


Figure XIV.3: Graph of the macro "dataserverLoadASCIIFileYoungsModulus.C"

XIV.2.5.4 Console

```
Processing loadASCIIFileYoungsModulus.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

*******************************
** TAttribute::printLog  ******
**    Name[E]
**   Title[ Young's Modulues]
**   unity[ksi]
**    type[0]
**   share[1]
**   Origin[kAttribute]
**   Attribute[kInput]
**   _snote[]
---------------------------------------------------------------------------------------------- ↩

---------------------------------------------------------------------------------------------- ↩

** - min[25900] max[33400]
** - mean[29575.6] std[1506.95]
**   lowerBound[-1.42387e+64] upperBound[1.42387e+64]
** NOT _defaultValue[]
** NOT _stepValue[]
 ** No Attribute to substitute level[0]
*******************************
```

XIV.2.6 Macro "dataserverLoadASCIIFileIonosphere.C"

XIV.2.6.1 Objective

The objective of this macro is to load the ASCII data file `ionosphere.dat` which defines 34 input variables and one output variable **y** and applies visualisation on one of these variables. The data file `ionosphere.dat` is in the "Salome-table" format of Uranie but is not shown for convenience.

XIV.2.6.2 Macro Uranie

```
{
  TDataServer * tds = new TDataServer();
  tds->fileDataRead("ionosphere.dat");

  tds->getAttribute("x28")->SetTitle("#Delta P_{e}^{F_{iso}}");

  // Graph
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro loadASCIIFileIonosphere" ↩
      ,5,64,1270,667);
  tds->draw("x28");

}
```

The `TDataServer` is filled with `ionosphere.dat` with the `fileDataRead` method

```
tds->fileDataRead("ionosphere.dat");
```

A new title is set for the variable **x28**

```
tds->getAttribute("x28")->SetTitle("#Delta P_{e}^{F_{iso}}");
```

This variable is then drawn with its new title

```
tds->draw("x28");
```

XIV.2.6.3 Graph

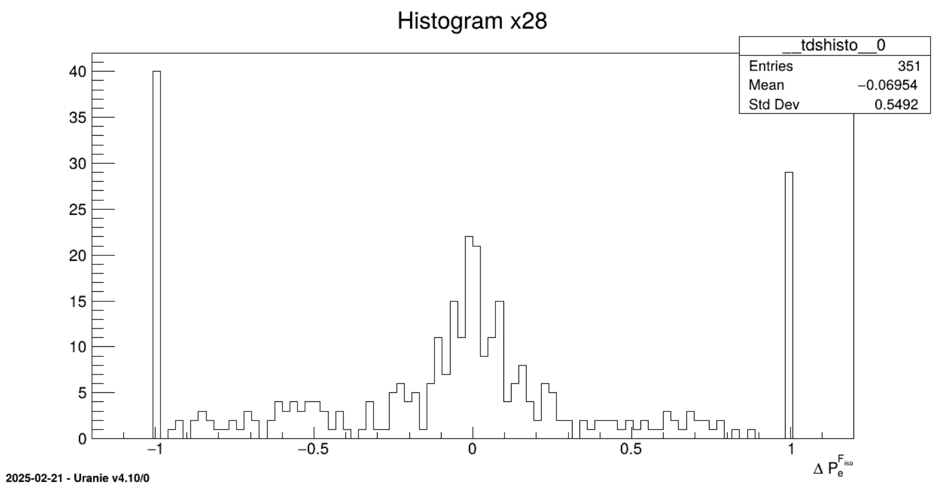


Figure XIV.4: Graph of the macro "dataserverLoadASCIIFileIonosphere.C"

XIV.2.7 Macro "dataserverLoadASCIIFileCornell.C"

XIV.2.7.1 Objective

The objective of this macro is to load the ASCII data file `cornell.dat` which defines seven input variables and one output variable **y** on twelve patterns. The input file `cornell.dat` is in the "Salome-table" format of Uranie

```
#NAME: cornell
#TITLE: Dataset Cornell 1990
#COLUMN_NAMES: x1 | x2 | x3 | x4 | x5 | x6 | x7 | y
#COLUMN_TITLES: x_{1} | x_{2} | x_{3} | x_{4} | x_{5} | x_{6} | x_{7} | y

0.00 0.23 0.00 0.00 0.00 0.74 0.03 98.7
0.00 0.10 0.00 0.00 0.12 0.74 0.04 97.8
0.00 0.00 0.00 0.10 0.12 0.74 0.04 96.6
0.00 0.49 0.00 0.00 0.12 0.37 0.02 92.0
0.00 0.00 0.00 0.62 0.12 0.18 0.08 86.6
0.00 0.62 0.00 0.00 0.00 0.37 0.01 91.2
0.17 0.27 0.10 0.38 0.00 0.00 0.08 81.9
0.17 0.19 0.10 0.38 0.02 0.06 0.08 83.1
0.17 0.21 0.10 0.38 0.00 0.06 0.08 82.4
0.17 0.15 0.10 0.38 0.02 0.10 0.08 83.2
0.21 0.36 0.12 0.25 0.00 0.00 0.06 81.4
0.00 0.00 0.00 0.55 0.00 0.37 0.08 88.1
```

XIV.2.7.2 Macro Uranie

```
{

  TDataServer * tds = new TDataServer();
  tds->fileDataRead("cornell.dat");

  TMatrix matCorr = tds->computeCorrelationMatrix("");
  matCorr.Print();
}
```

The TDataServer is filled with cornell.dat with the fileDataRead method:

```
tds->fileDataRead("cornell.dat");
```

Then the correlation matrix is computed on all attributes:

```
TMatrix matCorr = tds->computeCorrelationMatrix("");
```

XIV.2.7.3 Console

```
Processing loadASCIIFileCornell.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025


8x8 matrix is as follows

     |      0    |      1    |      2    |      3    |      4    |
----------------------------------------------------------------------
   0 |          1      0.1042      0.9999      0.3707      -0.548
   1 |     0.1042           1      0.1008     -0.5369     -0.2926
   2 |     0.9999      0.1008           1       0.374     -0.5482
   3 |     0.3707     -0.5369       0.374           1     -0.2113
   4 |     -0.548     -0.2926     -0.5482     -0.2113           1
   5 |    -0.8046     -0.1912     -0.8052     -0.6457      0.4629
   6 |     0.6026       -0.59      0.6071      0.9159     -0.2744
   7 |    -0.8373    -0.07082      -0.838     -0.7067      0.4938


     |      5    |      6    |      7    |
----------------------------------------------------------------------
   0 |    -0.8046      0.6026     -0.8373
   1 |    -0.1912       -0.59    -0.07082
   2 |    -0.8052      0.6071      -0.838
   3 |    -0.6457      0.9159     -0.7067
   4 |     0.4629     -0.2744      0.4938
   5 |          1     -0.6564      0.9851
   6 |    -0.6564           1     -0.7411
   7 |     0.9851     -0.7411           1
```

XIV.2.8 Macro "dataserverComputeQuantile.C"

XIV.2.8.1 Objective

The objective of this macro is to test the classical quantile estimation and compare it to the Wilks estimation for a dummy gaussian distribution. Four different estimations of the 95% quantile value are done (along with one estimation of th 99% quantile) for illustration purposes:

- using the usual method with a 200-points sample.

- using the usual method with a 400-points sample.
- using the Wilks method with a 95% confidence level (with 59-points sample).
- using the Wilks method with a 95% confidence level (with 400-points sample).
- using the Wilks method with a 99% confidence level (with 90-points sample).

XIV.2.8.2 Macro

```
{

 //Create a DataServer
 TDataServer *tds = new TDataServer("foo","pouet");
 tds->addAttribute("x"); //With one attribute

 //Create Histogram to store the quantile values
 TH1F *Q200 = new TH1F("quantile200","",60,1,4); Q200->SetLineColor(1); Q200->SetLineWidth ↩
     (2);
 TH1F *Q400 = new TH1F("quantile400","",60,1,4); Q400->SetLineColor(4); Q400->SetLineWidth ↩
     (2);
 TH1F *QW95 = new TH1F("quantileWilks95","",60,1,4); QW95->SetLineColor(2); QW95-> ↩
     SetLineWidth(2);
 TH1F *QW95400 = new TH1F("quantileWilks95400","",60,1,4); QW95400->SetLineColor(8);  ↩
     QW95400->SetLineWidth(2);
 TH1F *QW99 = new TH1F("quantileWilks99","",60,1,4); QW99->SetLineColor(6); QW99-> ↩
     SetLineWidth(2);

 //Defining the sample size
 int nb[4] = {200,400,59,90};
 double quant=0.95; //Quantile value
 double CL[2] = {0.95, 0.99};//Confidence level value for the two Wilks computation
 //Loop over the number of estimation (2 usual with different number of sample and 1 with  ↩
     Wilks estimation)
 for(unsigned int iq=0; iq<4; iq++)
 {
     //Produce 10000 drawing to get smooth distributio,
     for(unsigned int itest=0; itest < 10000; itest++)
     {
         tds->createTuple(); // Create the tuple to store value

   // Fill it with random drawing of centered gaussian
   for(unsigned int ival=0; ival < nb[iq]; ival++)
       tds->getTuple()->Fill(ival+1, gRandom->Gaus(0,1) );

   // Estimate the quantile...
   double value;
   if (iq<2)
   {
       // ... with usual methods...
       tds->computeQuantile("x",quant, value);
       if(iq==0) Q200->Fill(value); // ... on a 200-points sample
       else
       {
           Q400->Fill(value); // ... on a 400-points sample
     tds->estimateQuantile("x", quant, value, CL[iq-1]);
     QW95400->Fill(value); // compute the quantile at 95% CL
       }
   }
   else
```

```
    {
        // ... with the wilks optimised sample
        tds->estimateQuantile("x", quant, value, CL[iq-2]);
        if(iq==2) QW95->Fill(value); // compute the quantile at 95% CL
        else QW99->Fill(value); // compute the quantile at 99% CL
    }

    // Delete the tuple
    tds->deleteTuple();
      }
   }

   //Produce the plot with requested style
   gStyle->SetOptStat(0);
   TCanvas *can = new TCanvas("Can","Can",10,10,1000,1000);
   Q400->GetXaxis()->SetTitle("Quant_{95%}(Gaus_{(0,1)})");
   Q400->GetXaxis()->SetTitleOffset(1.2);
   Q400->Draw();
   Q200->Draw("same");
   QW95->Draw("same");
   QW95400->Draw("same");
   QW99->Draw("same");

   //Add the theoretical estimation
   TLine *lin = new TLine(); lin->SetLineStyle(3);
   lin->DrawLine(1.645,0,1.645, Q400->GetMaximum());

   //Add a block of legend
   TLegend *leg = new TLegend(0.4,0.6,0.8,0.85);
   leg->AddEntry(lin,"Theoretical quantile","l");
   leg->AddEntry(Q200, "Usual quantile (200 pts)","l");
   leg->AddEntry(Q400, "Usual quantile (400 pts)","l");
   leg->AddEntry(QW95, "Wilks quantile CL=95% (59 pts)","l");
   leg->AddEntry(QW95400, "Wilks quantile CL=95% (400 pts)","l");
   leg->AddEntry(QW99, "Wilks quantile CL=99% (90 pts)","l");
   leg->SetBorderSize(0);
   leg->Draw();


}
```

In this macro, a dummy dataserver is created with a single attribute named "x". Four histograms are prepared to store the resulting value. Then the same loop will be used to computed 10000 values of every quantile with different switchs to use one method instead of the other, or to change the number of points in the sample and/or the confidence level. All this is defined in the small part before the loop:

```
//Defining the sample size
int nb[4] = {200,400,59,90};
double quant=0.95; //Quantile value
double CL[2] = {0.95, 0.99};//Confidence level value for the two Wilks computation
```

Then the computation is performed, first for the usual method (first two iterations of `iq`), then for the Wilks estimation (last two iterations of `iq`). Every computational result is stored in the corresponding histogram which is finally displayed and shown in the following subsection.

XIV.2.8.3 Graph

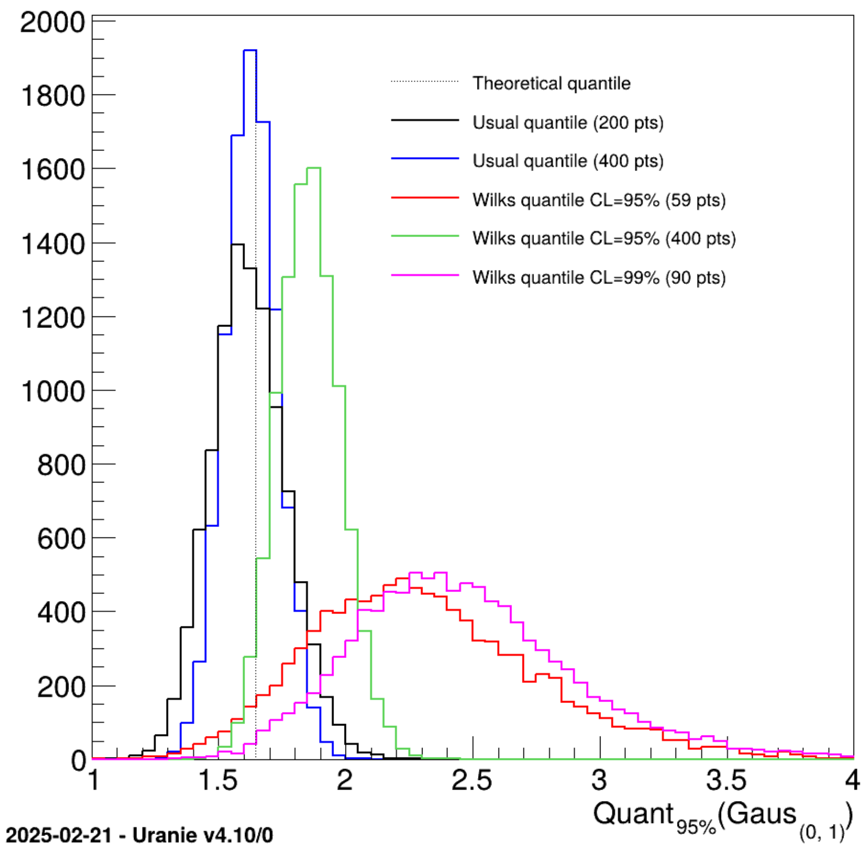


Figure XIV.5: Graph of the macro "`dataserverComputeQuantile.C`"

XIV.2.9 Macro "`dataserverGeyserStat.C`"

XIV.2.9.1 Objective

This part shows the complete code used to produce the console display in Section II.4.3.

XIV.2.9.2 Macro Uranie

```
{
  TDataServer *tdsGeyser =new TDataServer("geyser","poet");
  tdsGeyser->fileDataRead("geyser.dat");
  tdsGeyser->computeStatistic("x1");

  cout<<"min(x1)= "<<tdsGeyser->getAttribute("x1")->getMinimum()<<";  max(x1)= "<<tdsGeyser ↩
      ->getAttribute("x1")->getMaximum()
      <<";  mean(x1)= "<<tdsGeyser->getAttribute("x1")->getMean()<<";  std(x1)= "<< ↩
          tdsGeyser->getAttribute("x1")->getStd()<<endl;
  }
```

XIV.2.9.3 Console

This macro should result in this output in console:

```
min(x1)= 1.6;  max(x1)= 5.1;  mean(x1)= 3.48778;  std(x1)= 1.14137
```

XIV.2.10 Macro "dataserverGeyserRank.C"

XIV.2.10.1 Objective

This part shows the complete code used to produce the console display in Section II.4.2.

XIV.2.10.2 Macro Uranie

```
{
  TDataServer *tdsGeyser =new TDataServer("geyser","poet");
  tdsGeyser->fileDataRead("geyser.dat");
  tdsGeyser->computeRank("x1");
  tdsGeyser->computeStatistic("Rk_x1");

  cout<<"NPatterns="<<tdsGeyser->getNPatterns()<<";  min(Rk_x1)= "<<tdsGeyser->getAttribute ↩
      ("Rk_x1")->getMinimum()
      <<";  max(Rk_x1)= "<<tdsGeyser->getAttribute("Rk_x1")->getMaximum()<<endl;

  }
```

XIV.2.10.3 Console

This macro should result in this output in console:

```
NPatterns=272;  min(Rk_x1)= 1;  max(Rk_x1)= 272
```

XIV.2.11 Macro "dataserverNormaliseVector.C"

XIV.2.11.1 Objective

This part shows the complete code used to produce the console display in Section II.4.1.

XIV.2.11.2 Macro Uranie

```
{
  TDataServer *tdsop =new TDataServer("foo","pouet");
  tdsop->fileDataRead("tdstest.dat");

  //Compute a global normalisation of v, CenterReduced
  tdsop->normalize("v","GCR",TDataServer::kCR,true);
  //Compute a normalisation of v, CenterReduced (not global but entry by entry)
  tdsop->normalize("v","CR",TDataServer::kCR,false);

  //Compute a global normalisation of v, Centered
  tdsop->normalize("v","GCent",TDataServer::kCentered);
  //Compute a normalisation of v, Centered  (not global but entry by entry)
  tdsop->normalize("v","Cent",TDataServer::kCentered,false);

  //Compute a global normalisation of v, ZeroOne
  tdsop->normalize("v","GZO",TDataServer::kZeroOne);
  //Compute a normalisation of v, ZeroOne (not global but entry by entry)
  tdsop->normalize("v","ZO",TDataServer::kZeroOne,false);

  //Compute a global normalisation of v, MinusOneOne
  tdsop->normalize("v","GMOO",TDataServer::kMinusOneOne,true);
  //Compute a normalisation of v, MinusOneOne (not global but entry by entry)
  tdsop->normalize("v","MOO",TDataServer::kMinusOneOne,false);

  tdsop->scan("v:vGCR:vCR:vGCent:vCent:vGZO:vZO:vGMOO:vMOO","","colsize=4 col=2:5:::::::") ↩
      ;
  }
```

XIV.2.11.3 Console

This macro should result in this output in console:

```
************************************************************************************
*    Row   * Instance *  v *  vGCR *  vCR * vGCe * vCen * vGZO *  vZO * vGMO * vMOO *
************************************************************************************
*        0 *        0 *  1 * -1.46 *   -1 *   -4 *   -3 *    0 *    0 *   -1 *   -1 *
*        0 *        1 *  2 * -1.09 *   -1 *   -3 *   -3 * 0.12 *    0 * -0.7 *   -1 *
*        0 *        2 *  3 * -0.73 *   -1 *   -2 *   -3 * 0.25 *    0 * -0.5 *   -1 *
*        1 *        0 *  4 * -0.36 *    0 *   -1 *    0 * 0.37 *  0.5 * -0.2 *    0 *
*        1 *        1 *  5 *     0 *    0 *    0 *    0 *  0.5 *  0.5 *    0 *    0 *
*        1 *        2 *  6 * 0.365 *    0 *    1 *    0 * 0.62 *  0.5 * 0.25 *    0 *
*        2 *        0 *  7 * 0.730 *    1 *    2 *    3 * 0.75 *    1 *  0.5 *    1 *
*        2 *        1 *  8 * 1.095 *    1 *    3 *    3 * 0.87 *    1 * 0.75 *    1 *
*        2 *        2 *  9 * 1.460 *    1 *    4 *    3 *    1 *    1 *    1 *    1 *
************************************************************************************
```

XIV.2.12 Macro "dataserverComputeStatVector.C"

XIV.2.12.1 Objective

This part shows the complete code used to produce the console display in Section II.4.3.1.

XIV.2.12.2 Macro Uranie

```
{
  TDataServer *tdsop =new TDataServer("foo","poet");
  tdsop->fileDataRead("tdstest.dat");

  //Considering every element of a vector independent from the others
  tdsop->computeStatistic("x");
  TAttribute *px = tdsop->getAttribute("x");

  cout<<"min(x[0])= "<<px->getMinimum(0)<<";  max(x[0])= "<<px->getMaximum(0)
      <<";  mean(x[0])= "<<px->getMean(0)<<";  std(x[0])= "<<px->getStd(0)<<endl;
  cout<<"min(x[1])= "<<px->getMinimum(1)<<";  max(x[1])= "<<px->getMaximum(1)
      <<";  mean(x[1])= "<<px->getMean(1)<<";  std(x[1])= "<<px->getStd(1)<<endl;
  cout<<"min(x[2])= "<<px->getMinimum(2)<<";  max(x[2])= "<<px->getMaximum(2)
      <<";  mean(x[2])= "<<px->getMean(2)<<";  std(x[2])= "<<px->getStd(2)<<endl;
  cout<<"min(xtot)= "<<px->getMinimum(3)<<";  max(xtot)= "<<px->getMaximum(3)
      <<";  mean(xtot)= "<<px->getMean(3)<<";  std(xtot)= "<<px->getStd(3)<<endl;

  //Statistic for a single realisation of a vector, not considering other events
  tdsop->addAttribute("Min_x","Min$(x)");
  tdsop->addAttribute("Max_x","Max$(x)");
  tdsop->addAttribute("Mean_x","Sum$(x)/Length$(x)");

  tdsop->scan("x:Min_x:Max_x:Mean_x","","colsize=5 col=2::::");
  }
```

XIV.2.12.3 Console

This macro should result in this output in console, split in two parts, the first one being from Uranie's method

```
min(x[0])= 1;  max(x[0])= 7;  mean(x[0])= 3;  std(x[0])= 3.4641
min(x[1])= 2;  max(x[1])= 8;  mean(x[1])= 4.66667;  std(x[1])= 3.05505
min(x[2])= 3;  max(x[2])= 9;  mean(x[2])= 6.66667;  std(x[2])= 3.21455
min(xtot)= 1;  max(xtot)= 9;  mean(xtot)= 4.77778;  std(xtot)= 3.23179
```

The second on the other hand results from ROOT's methods (the second part of the code shown above):

```
*****************************************************
*    Row   * Instance *   x * Min_x * Max_x * Mean_x *
*****************************************************
*        0 *        0 *   1 *     1 *     3 *      2 *
*        0 *        1 *   2 *     1 *     3 *      2 *
*        0 *        2 *   3 *     1 *     3 *      2 *
*        1 *        0 *   7 *     7 *     9 *      8 *
*        1 *        1 *   8 *     7 *     9 *      8 *
*        1 *        2 *   9 *     7 *     9 *      8 *
*        2 *        0 *   1 *     1 *     8 * 4.3333 *
*        2 *        1 *   4 *     1 *     8 * 4.3333 *
*        2 *        2 *   8 *     1 *     8 * 4.3333 *
*****************************************************
```

XIV.2.13 Macro "dataserverComputeCorrelationMatrixVector.C"

XIV.2.13.1 Objective

This part shows the complete code used to produce the console display in Section II.4.5.1.

XIV.2.13.2 Macro Uranie

```
{
  TDataServer *tdsop =new TDataServer("foo","poet");
  tdsop->fileDataRead("tdstest.dat");

  // Consider a and x attributes (every element of the vector)
  TMatrixD globalOne = tdsop->computeCorrelationMatrix("x:a");
  globalOne.Print();

  // Consider a and x attributes (cherry-picking a single element of the vector)
  TMatrixD focusedOne = tdsop->computeCorrelationMatrix("x[1]:a");
  focusedOne.Print();
}
```

XIV.2.13.3 Console

This macro should result in this output in console.

```
4x4 matrix is as follows

     |      0    |      1    |      2    |      3    |
----------------------------------------------------------
   0 |          1      0.9449      0.6286       0.189
   1 |     0.9449           1      0.8486         0.5
   2 |     0.6286      0.8486           1      0.8825
   3 |      0.189         0.5      0.8825           1


2x2 matrix is as follows

     |      0    |      1    |
-------------------------------
   0 |          1         0.5
   1 |        0.5           1
```

XIV.2.14 Macro "dataserverComputeQuantileVec.C"

XIV.2.14.1 Objective

This part shows the complete code used to produce the console display in Section II.4.4.1.

XIV.2.14.2 Macro Uranie

```
{
    TDataServer *tdsvec = new TDataServer("foo", "bar");
    tdsvec->fileDataRead("aTDSWithVectors.dat");

    double probas[3]={0.2, 0.6, 0.8}; double quants[3];
    tdsvec->computeQuantile("rank", 3, probas, quants);

    TAttribute *prank = tdsvec->getAttribute("rank");
    int nbquant;
    prank->getQuantilesSize(nbquant); // (1)
    cout << "nbquant = " << nbquant << endl;

    double aproba=0.8; double aquant;
    prank->getQuantile(aproba, aquant); // (2)
    cout << "aproba = " << aproba << ", aquant = " <<
    aquant << endl;

    double theproba[3], thequant[3];
    prank->getQuantiles(theproba, thequant); // (3)
    for(int i_quant=0; i_quant<nbquant; ++i_quant) {
        cout << "(theproba, thequant)[" << i_quant << "] = "
        << "(" << theproba[i_quant] << ", " <<
        thequant[i_quant] << ")" << endl;
    }

    vector<double> allquant;
    prank->getQuantileVector(aproba, allquant); // (4)
    cout << "aproba = " << aproba << ", allquant = ";
    for(double quant_i: allquant)
        cout << quant_i << " ";
    cout << endl;
}
```

XIV.2.14.3 Console

This macro should result in this output in console:

```
nbquant = 3
aproba = 0.8, aquant = 6.4
(theproba, thequant)[0] = (0.2, 1.6)
(theproba, thequant)[1] = (0.6, 4.8)
(theproba, thequant)[2] = (0.8, 6.4)
aproba = 0.8, allquant = 6.4 7.4
```

XIV.2.15 Macro "dataserverDrawQQPlot.C"

XIV.2.15.1 Objective

This macro is an example of how to produce QQ-plot for a certain number of randomly-drawn samples, providing the correct parameter values along with modified versions to illustrate the impact.

XIV.2.15.2 Macro Uranie

```
{
    // Create a TDS with 8 kind of distributions
    double p1=1.3, p2=4.5, p3=0.9, p4=4.4; // Fixed values for parameters

    TDataServer *tds0 = new TDataServer();
    tds0->addAttribute(new TNormalDistribution("norm", p1, p2));
    tds0->addAttribute(new TLogNormalDistribution("logn", p1, p2));
    tds0->addAttribute(new TUniformDistribution("unif", p1, p2));
    tds0->addAttribute(new TExponentialDistribution("expo", p1, p2));
    tds0->addAttribute(new TGammaDistribution("gamm", p1, p2, p3));
    tds0->addAttribute(new TBetaDistribution("beta", p1, p2, p3, p4));
    tds0->addAttribute(new TWeibullDistribution("weib", p1, p2, p3));
    tds0->addAttribute(new TGumbelMaxDistribution("gumb", p1, p2));

    // Create the sample
    TBasicSampling *fsamp = new TBasicSampling(tds0, "lhs", 200);
    fsamp->generateSample();

    // Define number of laws, their name and numbers of parameters
    unsigned int nLaws=8;
    string laws[8]={"normal", "lognormal", "uniform", "gamma", "weibull", "beta", " ↩
       exponential", "gumbelmax"}; // number of parameters to put in () for the ↩
       corresponding law
    int npar[8]={2, 2, 2, 3, 3, 4, 2, 2};

    //Create the canvas
    TCanvas *c = new TCanvas("c1","",800,1000);
    //Create the 8 pads
    TPad *apad = new TPad("apad","apad",0, 0.03, 1, 1); apad->Draw(); apad->cd();
    apad->Divide(2,4);

    // Number of points to compare theoretical and empirical values
    int nS=1000;
    double mod=0.8; // Factor used to artificially change the parameter values

    TString opt=""; //option of the drawQQPlot method
    stringstream sstr;
    for(unsigned int ilaw=0; ilaw<nLaws; ilaw++)
    {
        // Clean sstr
        sstr.str("");
        // Add nominal configuration
        sstr << laws[ilaw] << "("<<p1<<","<<p2<<((npar[ilaw]>=3)?Form(",%g",p3):"")<<((npar ↩
            [ilaw]>=4)?Form(",%g",p4):"")<<")";
        // Changing par1
        sstr << ":" << laws[ilaw] << "("<<p1*mod<<","<<p2<<((npar[ilaw]>=3)?Form(",%g",p3): ↩
            "")<<((npar[ilaw]>=4)?Form(",%g",p4):"")<<")";
        // Changing par2
        sstr << ":" << laws[ilaw] << "("<<p1<<","<<p2*mod<<((npar[ilaw]>=3)?Form(",%g",p3): ↩
            "")<<((npar[ilaw]>=4)?Form(",%g",p4):"")<<")";
        // Changing par3
        if(npar[ilaw] >=3 )
            sstr << ":" << laws[ilaw] << "("<<p1<<","<<p2<<((npar[ilaw]>=3)?Form(",%g",p3* ↩
                mod):"")<<((npar[ilaw]>=4)?Form(",%g",p4):"")<<")";
        // Changing par4
        if(npar[ilaw] >=4 )
            sstr << ":" << laws[ilaw] << "("<<p1<<","<<p2<<((npar[ilaw]>=3)?Form(",%g",p3): ↩
                "")<<((npar[ilaw]>=4)?Form(",%g",p4*mod):"")<<")";
        //cout<<sstr.str()<<endl;

        apad->cd(ilaw+1);
        // Produce the plot
        tds0->drawQQPlot( laws[ilaw].substr(0,4).c_str(), sstr.str().c_str(), nS, opt);
    }

}
```

The very first step of this macro is to create a sample that will contain a design-of-experiments filled with 200 locations, using various statistical laws. All the tested laws, are those available in the `drawQQPlot` method and they might depend on 2 to 4 parameters, defined a but randomly at the beginning of this piece of code.

```
// Create a TDS with 8 kind of distributions
double p1=1.3, p2=4.5, p3=0.9, p4=4.4; // Fixed values for parameters

TDataServer *tds0 = new TDataServer();
tds0->addAttribute(new TNormalDistribution("norm", p1, p2));
tds0->addAttribute(new TLogNormalDistribution("logn", p1, p2));
tds0->addAttribute(new TUniformDistribution("unif", p1, p2));
tds0->addAttribute(new TExponentialDistribution("expo", p1, p2));
tds0->addAttribute(new TGammaDistribution("gamm", p1, p2, p3));
tds0->addAttribute(new TBetaDistribution("beta", p1, p2, p3, p4));
tds0->addAttribute(new TWeibullDistribution("weib", p1, p2, p3));
tds0->addAttribute(new TGumbelMaxDistribution("gumb", p1, p2));
```

Once done, the sample is generated using `TBasicSampling` object with an LHS algorithm. On top of this, despite the plot preparation with canvas and pad generation, several variables are set to prepare the tests, as shown below

```
// Define number of laws, their name and numbers of parameters
unsigned int nLaws=8;
string laws[8]={"normal", "lognormal", "uniform", "gamma", "weibull", "beta", "exponential" ↩
    , "gumbelmax"};
int npar[8]={2, 2, 2, 3, 3, 4, 2, 2}; // number of parameters to put in () for the  ↩
    corresponding law

// Number of points to compare theoretical and empirical values
int nS=1000;
double mod=0.8; // Factor used to artificially change the parameter values
```

Finally, after the line of hypothesis to be tested is constructed (the first paragraph in the for loop) the `drawQQPlot` method is called for every empirical law in the following line.

```
 tds0->drawQQPlot( laws[ilaw].substr(0,4).c_str(), sstr.str().c_str(), nS, opt);
```

For the first case, when one wants to test the `TNormalDistribution` "norm" with the known parameters and a variation of each, it resumes as if this line was run:

```
tds0->drawQQPlot( "norm", "normal(1.3,4.5):normal(1.04,4.5):normal(1.3,3.6)", nS);
```

The first field is the attribute to be tested, while the second one provides the three hypothesis with which our attribute under investigation will be compared. The third argument is the number of steps to be computed for quantiles. The result of this macro is shown below.

XIV.2.15.3 Graph

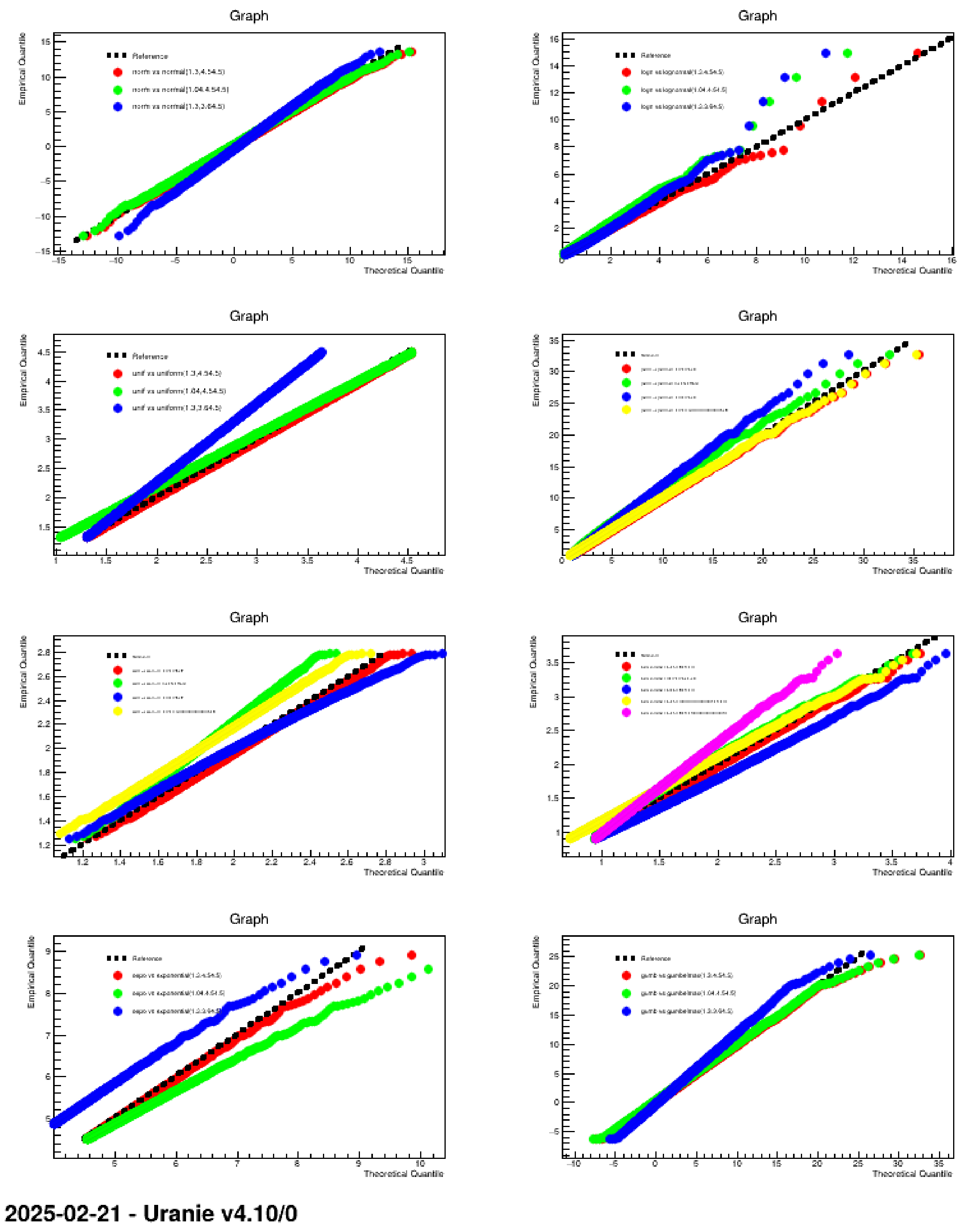


Figure XIV.6: Graph of the macro "`dataserverDrawQQPlot.C`"

XIV.2.16 Macro "dataserverDrawPPPlot.C"

XIV.2.16.1 Objective

This macro is an example of how to produce PP-plot for a certain number of randomly-drawn samples, providing the correct parameter values along with modified versions to illustrate the impact.

XIV.2.16.2 Macro Uranie

```
{
    // Create a TDS with 8 kind of distributions
    double p1=1.3, p2=4.5, p3=0.9, p4=4.4; // Fixed values for parameters

    TDataServer *tds0 = new TDataServer();
    tds0->addAttribute(new TNormalDistribution("norm", p1, p2));
    tds0->addAttribute(new TLogNormalDistribution("logn", p1, p2));
    tds0->addAttribute(new TUniformDistribution("unif", p1, p2));
    tds0->addAttribute(new TExponentialDistribution("expo", p1, p2));
    tds0->addAttribute(new TGammaDistribution("gamm", p1, p2, p3));
    tds0->addAttribute(new TBetaDistribution("beta", p1, p2, p3, p4));
    tds0->addAttribute(new TWeibullDistribution("weib", p1, p2, p3));
    tds0->addAttribute(new TGumbelMaxDistribution("gumb", p1, p2));

    // Create the sample
    TBasicSampling *fsamp = new TBasicSampling(tds0, "lhs", 200);
    fsamp->generateSample();

    // Define number of laws, their name and numbers of parameters
    unsigned int nLaws=8;
    string laws[8]={"normal", "lognormal", "uniform", "gamma", "weibull", "beta", " ↩
        exponential", "gumbelmax"}; // number of parameters to put in () for the ↩
        corresponding law
    int npar[8]={2, 2, 2, 3, 3, 4, 2, 2};

    //Create the canvas
    TCanvas *c = new TCanvas("c1","",800,1000);
    //Create the 8 pads
    TPad *apad = new TPad("apad","apad",0, 0.03, 1, 1); apad->Draw(); apad->cd();
    apad->Divide(2,4);

    // Number of points to compare theoretical and empirical values
    int nS=1000;
    double mod=0.8; // Factor used to artificially change the parameter values

    TString opt=""; //option of the drawPPPlot method
    stringstream sstr;
    for(unsigned int ilaw=0; ilaw<nLaws; ilaw++)
    {
        // Clean sstr
        sstr.str("");
        // Add nominal configuration
        sstr << laws[ilaw] << "("<<p1<<","<<p2<<((npar[ilaw]>=3)?Form(",%g",p3):"")<<((npar ↩
            [ilaw]>=4)?Form(",%g",p4):"")<<")";
        // Changing par1
        sstr << ":" << laws[ilaw] << "("<<p1*mod<<","<<p2<<((npar[ilaw]>=3)?Form(",%g",p3): ↩
            "")<<((npar[ilaw]>=4)?Form(",%g",p4):"")<<")";
        // Changing par2
        sstr << ":" << laws[ilaw] << "("<<p1<<","<<p2*mod<<((npar[ilaw]>=3)?Form(",%g",p3): ↩
            "")<<((npar[ilaw]>=4)?Form(",%g",p4):"")<<")";
        // Changing par3
        if(npar[ilaw] >=3 )
            sstr << ":" << laws[ilaw] << "("<<p1<<","<<p2<<((npar[ilaw]>=3)?Form(",%g",p3* ↩
                mod):"")<<((npar[ilaw]>=4)?Form(",%g",p4):"")<<")";
        // Changing par4
        if(npar[ilaw] >=4 )
            sstr << ":" << laws[ilaw] << "("<<p1<<","<<p2<<((npar[ilaw]>=3)?Form(",%g",p3): ↩
                "")<<((npar[ilaw]>=4)?Form(",%g",p4*mod):"")<<")";
        cout<<sstr.str()<<endl;

        apad->cd(ilaw+1);
        // Produce the plot
        tds0->drawPPPlot( laws[ilaw].substr(0,4).c_str(), sstr.str().c_str(), nS, opt);
    }

}
```

The macro is based on the one discussed in Section XIV.2.15. The only difference is this line

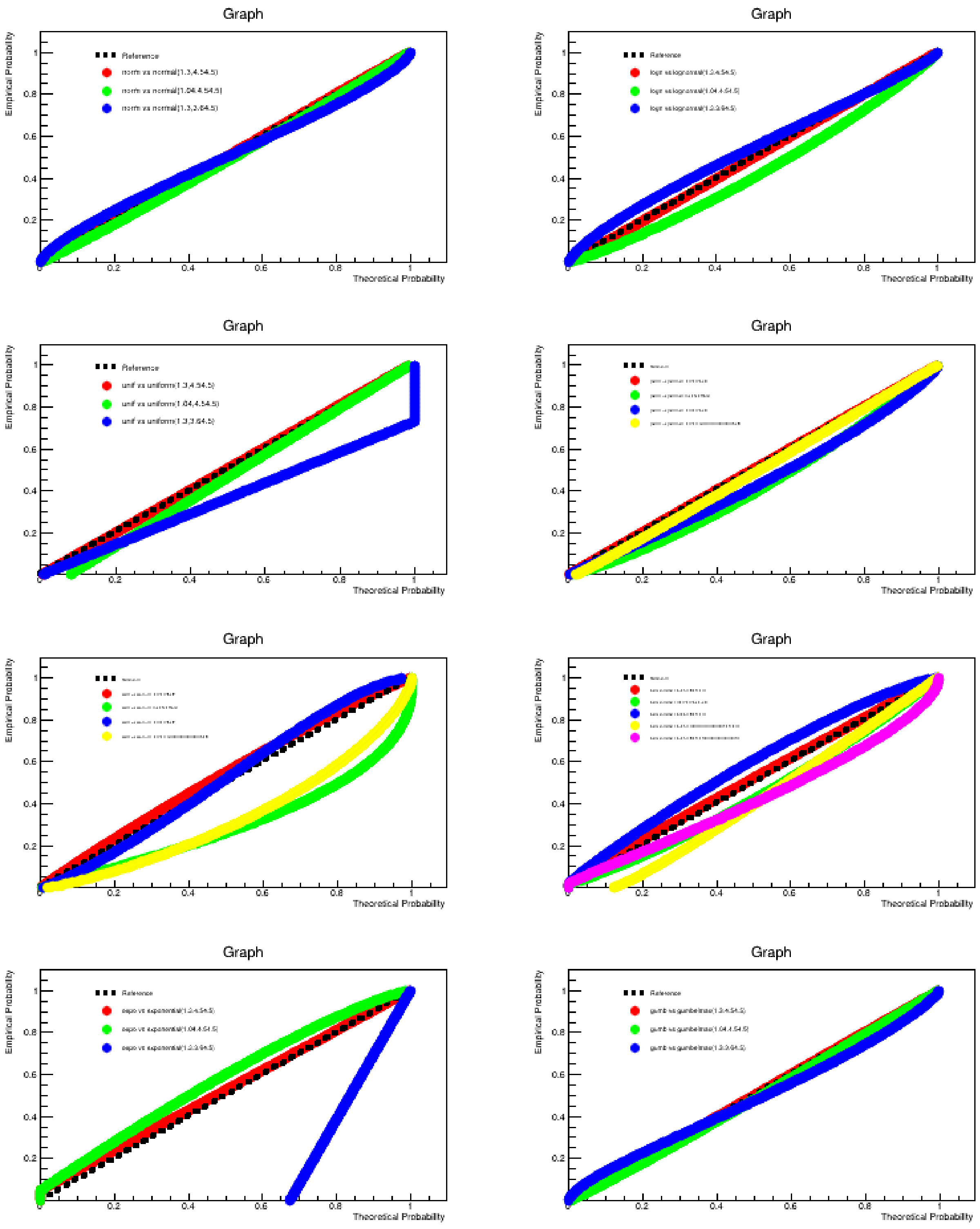
```
tds0->drawPPPlot( laws[ilaw].substr(0,4).c_str(), sstr.str().c_str(), nS, opt);
```

The call of the drawing method above can be resume, for the first case, like this:

```
tds0->drawPPPlot( "norm", "normal(1.3,4.5):normal(1.04,4.5):normal(1.3,3.6)", nS);
```

The first field is the attribute to be tested, while the second one provides the three hypothesis with which our attribute under investigation will be compared. The third argument is the number of steps to be computed for probabilities. The result of this macro is shown below.

XIV.2.16.3 Graph



2025-02-21 - Uranie v4.10/0

Figure XIV.7: Graph of the macro "dataserverDrawPPPlot.C"

XIV.2.17 Macro "dataserverPCAExample.C"

XIV.2.17.1 Objective

The goal of this macro is to show how to handle a PCA analysis. It is not much discussed here, as a large description of both methods and concepts is done in Section II.6.

XIV.2.17.2 Macro Uranie

```
{
  // Read the database
  TDataServer * tdsPCA = new TDataServer("tdsPCA", "my TDS");
  tdsPCA->fileDataRead("Notes.dat");

  // Create the PCA object precising the variables of interest
  TPCA * tpca = new TPCA(tdsPCA, "Maths:Physics:French:Latin:Music");
  tpca->compute();

  bool graphical=true; // do graphs
  bool dumponscreen=true; //or dumping results
  bool showcoordinate=false; // show the coordinate of the points while dumping results

  if(graphical)
  {

      // Draw all point in PCA planes
      TCanvas *cPCA = new TCanvas("cpca", "PCA",800,800);
      TPad *apad1 = new TPad("apad1","apad1",0, 0.03, 1, 1); apad1->Draw(); apad1->cd();
      apad1->Divide(2,2);
      apad1->cd(1); tpca->drawPCA(1,2,"Pupil");
      apad1->cd(3); tpca->drawPCA(1,3,"Pupil");
      apad1->cd(4); tpca->drawPCA(2,3,"Pupil");

      // Draw all variable weight in PC definition
      TCanvas *cLoading = new TCanvas("cLoading", "Loading Plot",800,800);
      TPad *apad2 = new TPad("apad2","apad2",0, 0.03, 1, 1); apad2->Draw(); apad2->cd();
      apad2->Divide(2,2);
      apad2->cd(1); tpca->drawLoading(1,2);
      apad2->cd(3); tpca->drawLoading(1,3);
      apad2->cd(4); tpca->drawLoading(2,3);

      // Draw the eigen values in different normalisation
      TCanvas *c = new TCanvas("cEigenValues", "Eigen Values Plot",1100,500);
      TPad *apad3 = new TPad("apad3","apad3",0, 0.03, 1, 1); apad3->Draw(); apad3->cd();
      apad3->Divide(3,1);
      TNtupleD *ntd = tpca->getResultNtupleD();
      apad3->cd(1); ntd->Draw("eigen:i","","lp");
      apad3->cd(2); ntd->Draw("eigen_pct:i","","lp"); gPad->SetGrid();
      apad3->cd(3); ntd->Draw("sum_eigen_pct:i","","lp"); gPad->SetGrid();

  }

  if(dumponscreen)
  {

      int nPCused=5; // 3 to see only the meaningful ones
      TString PCname="", Cosname="", Contrname="", Variable="Pupil";
      for (unsigned int iatt=1; iatt<=nPCused; iatt++)
      {
          PCname+=Form("PC_%d",iatt)+((iatt!=nPCused)?TString(":"):TString(""));
          Cosname+=Form("cosca_%d",iatt)+((iatt!=nPCused)?TString(":"):TString(""));
          Contrname+=Form("contr_%d",iatt)+((iatt!=nPCused)?TString(":"):TString(""));
      }

      cout<<endl<<"========= EigenValues ====================="<<endl;
      tpca->getResultNtupleD()->Scan("*");

      if(showcoordinate)
      {
          cout<<endl<<"========= EigenVectors ====================="<<endl;
          tpca->_matEigenVectors.Print();

          cout<<endl<<"========= New Coordinates ====================="<<endl;
          tdsPCA->scan( (Variable+":"+PCname).Data() );
      }


      TDSNtupleD *varRes  = tpca->getVariableResultNtupleD();
      cout<<endl<<"=============== Looking at variables: Quality of representation ↩
          ====================="<<endl;
      varRes->Scan(("Variable:"+Cosname).Data());

      cout<<endl<<"==================== Looking at variables: Contribution to axis ↩
          ====================="<<endl;
      varRes->Scan(("Variable:"+Contrname).Data());

      cout<<endl<<"================== Looking at events:  Quality of representation ↩
          ===================="<<endl;
      tdsPCA->scan((Variable+":"+Cosname).Data());

      cout<<endl<<"===================== Looking at events: Contribution to axis ↩
          ======================="<<endl;
      tdsPCA->scan((Variable+":"+Contrname).Data());

   }
}
```

This first part of this macro is described in Section II.6. We will focus here on the second part, where all the numerical results are dumped on screen. These results are stored in the dataserver for the points and in a dedicated ntuple for the variable that can be retrieved by calling the method `getVariableResultNtupleD`. In both cases, the results can be split into two kinds:

- the quality of the representation: it is called *"cosca_X"* as it is a squared cosinus of the projection of the source under study (point or subject) on the X-th PC.
- the contribution to axis: it is called *"contr_X"* as it is the contribution of the source under study (point or subject) to the definition of the X-th PC.

We start by defining the list of variables that one might want to display

```
int nPCused=5; // 3 to see only the meaningful ones
TString PCname="", Cosname="", Contrname="", Variable="Pupil";
for (unsigned int iatt=1; iatt<=nPCused; iatt++)
{
//list of PC: PC_1:PC_2:PC_3:PC_4:PC_5
PCname+=Form("PC_%d",iatt)+((iatt!=nPCused)?TString(":"):TString(""));
//list of quality coeff: cosca_1:cosca_2:cosca_3:cosca_4:cosca_5
Cosname+=Form("cosca_%d",iatt)+((iatt!=nPCused)?TString(":"):TString(""));
//list of contribution: contr_1:contr_2:contr_3:contr_4:contr_5
Contrname+=Form("contr_%d",iatt)+((iatt!=nPCused)?TString(":"):TString(""));
}
```

From there, once the variable ntuple is retrieved, one can dump both the quality and contribution coefficients for the variable, here the subjects (it leads to the second and third block in the output shown in Section XIV.2.17.3).

```
TDSNtupleD *varRes  = tpca->getVariableResultNtupleD();
cout<<endl<<"=============== Looking at variables: Quality of representation ↩
    ====================="<<endl;
varRes->Scan(("Variable:"+Cosname).Data());

cout<<endl<<"==================== Looking at variables: Contribution to axis ↩
    ====================="<<endl;
varRes->Scan(("Variable:"+Contrname).Data());
```

Finally, one can do the same for the data points, with the same `Scan` method (which lead to the fourth and fifth block in the output shown in Section XIV.2.17.3).

```
cout<<endl<<"================== Looking at events:  Quality of representation ↩
    ===================="<<endl;
tdsPCA->scan((Variable+":"+Cosname).Data());

cout<<endl<<"===================== Looking at events: Contribution to axis ↩
    ======================="<<endl;
tdsPCA->scan((Variable+":"+Contrname).Data());
```

XIV.2.17.3 Console

```
========= EigenValues =====================
************************************************************
*    Row   *       i.i * eigen.eig * eigen_pct * sum_eigen *
************************************************************
*        0 *         1 * 2.8618175 * 57.236350 * 57.236350 *
*        1 *         2 * 1.1506811 * 23.013622 * 80.249973 *
*        2 *         3 * 0.9831407 * 19.662814 * 99.912787 *
*        3 *         4 * 0.0039371 * 0.0787424 * 99.991530 *
*        4 *         5 * 0.0004234 * 0.0084696 *       100 *
************************************************************

=============== Looking at variables: Quality of representation =====================
************************************************************************************
*    Row   * Variable *   cosca_1 *   cosca_2 *   cosca_3 *   cosca_4 *   cosca_5 *
************************************************************************************
*        0 *    Maths *  0.649485 *   0.32645 * 0.0235449 * 0.0003614 * 0.0001582 *
*        1 *  Physics *  0.804627 *  0.185581 * 0.0086212 * 0.0010515 * 0.0001193 *
*        2 *   French *  0.574697 *  0.373378 * 0.0509446 * 0.0008976 * 8.252e-05 *
*        3 *    Latin *  0.828562 *  0.157999 * 0.0117556 * 0.0016205 * 6.335e-05 *
*        4 *    Music * 0.0044470 *  0.107272 *  0.888274 * 5.976e-06 * 8.299e-08 *
************************************************************************************

==================== Looking at variables: Contribution to axis =====================
************************************************************************************
*    Row   * Variable *   contr_1 *   contr_2 *   contr_3 *   contr_4 *   contr_5 *
************************************************************************************
*        0 *    Maths *  0.226948 *  0.283702 * 0.0239486 * 0.0917987 *  0.373602 *
*        1 *  Physics *  0.281159 *   0.16128 * 0.0087691 *  0.267082 *   0.28171 *
*        2 *   French *  0.200815 *  0.324485 * 0.0518182 *  0.228004 *  0.194878 *
*        3 *    Latin *  0.289523 *  0.137309 * 0.0119572 *  0.411598 *  0.149613 *
*        4 *     Music * 0.0015539 * 0.0932252 *  0.903507 * 0.0015180 * 0.0001959 *
************************************************************************************

================== Looking at events:  Quality of representation ====================
************************************************************************************
*    Row   *      Pupil *    cosca_1 *    cosca_2 *    cosca_3 *    cosca_4 *    cosca_5 *
************************************************************************************
*        0 *       Jean * 0.8854534 * 0.0522119 * 0.0619429 * 0.0002655 * 0.0001260 *
*        1 *      Aline * 0.7920409 * 0.0542262 * 0.1530381 * 0.0006354 * 5.916e-05 *
*        2 *      Annie * 0.4784294 * 0.4813342 * 0.0384099 * 0.0018007 * 2.560e-05 *
*        3 *    Monique * 0.8785990 * 0.0024790 * 0.1180158 * 0.0009035 * 2.557e-06 *
*        4 *     Didier * 0.8515216 * 0.1382946 * 0.0079754 * 0.0021718 * 3.640e-05 *
*        5 *      Andre * 0.2465355 * 0.3961581 * 0.3567663 * 9.471e-05 * 0.0004451 *
*        6 *     Pierre * 0.0263090 * 0.7670958 * 0.2060832 * 0.0004625 * 4.925e-05 *
*        7 *   Brigitte * 0.1876629 * 0.5897686 * 0.2211409 * 0.0013390 * 8.836e-05 *
*        8 *    Evelyne * 0.0583185 * 0.3457931 * 0.5953786 * 0.0004600 * 4.957e-05 *
************************************************************************************

===================== Looking at events: Contribution to axis ======================
************************************************************************************
*    Row   *      Pupil *    contr_1 *    contr_2 *    contr_3 *    contr_4 *    contr_5 *
************************************************************************************
*        0 *       Jean * 0.3012931 * 0.0441856 * 0.0613538 * 0.0656820 * 0.2897335 *
*        1 *      Aline * 0.0618830 * 0.0105370 * 0.0348056 * 0.0360894 * 0.0312404 *
*        2 *      Annie * 0.0401366 * 0.1004284 * 0.0093797 * 0.1098075 * 0.0145176 *
*        3 *    Monique * 0.3784615 * 0.0026558 * 0.1479781 * 0.2828995 * 0.0074454 *
*        4 *     Didier * 0.1484067 * 0.0599446 * 0.0040461 * 0.2751439 * 0.0428726 *
*        5 *      Andre * 0.0348742 * 0.1393737 * 0.1469047 * 0.0097384 * 0.4255425 *
*        6 *     Pierre * 0.0041001 * 0.2973223 * 0.0934888 * 0.0524003 * 0.0518702 *
*        7 *   Brigitte * 0.0157710 * 0.1232678 * 0.0540974 * 0.0817989 * 0.0501849 *
*        8 *    Evelyne * 0.0150734 * 0.2222843 * 0.4479453 * 0.0864396 * 0.0865925 *
************************************************************************************
```

XIV.3 Macros Sampler

XIV.3.1 Macro "`samplingFlowrate.C`"

XIV.3.1.1 Objective

The objective of this macro is to generate a design-of-experiments, of length 100 using the LHS method, with eight random attributes (r_ω, r, T_u, T_l, H_u, H_l, L, K_ω) which obey uniform laws on specific intervals.

XIV.3.1.2 Macro Uranie

```
{
  Int_t nS = 1000;
  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsFlowrate", "Design of experiments for Flowrate");

  // Add the study attributes
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));


  // test Generateur de plan d'experience
  TSampling * sampling = new TSampling(tds, "lhs", nS);
  sampling->generateSample();

  tds->exportData("_flowrate_sampler_.dat");


  // Visualisation
  TCanvas  *Canvas = new TCanvas("Canvas", "Graph for the Macro samplingFlowrate" ↩
      ,5,64,1270,667);
  gStyle->SetPalette(1);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();

  pad->Divide(2, 2);
  pad->cd(1); tds->draw("r");
  pad->cd(2); tds->draw("rw");
  pad->cd(3); tds->drawTufte("rw:r");
  pad->cd(4); tds->draw("rw:r:hu");


}
```

An uniform law is set for each attribute and then, linked to a the `TDataServer`:

```
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

The sampling is generated with the LHS method:

```
TSampling * sampling = new TSampling(tds, "lhs", nS);
sampling->generateSample();
```

Data are exported in an ASCII file:

```
tds->exportData("_flowrate_sampler_.dat");
```

XIV.3.1.3 Graph

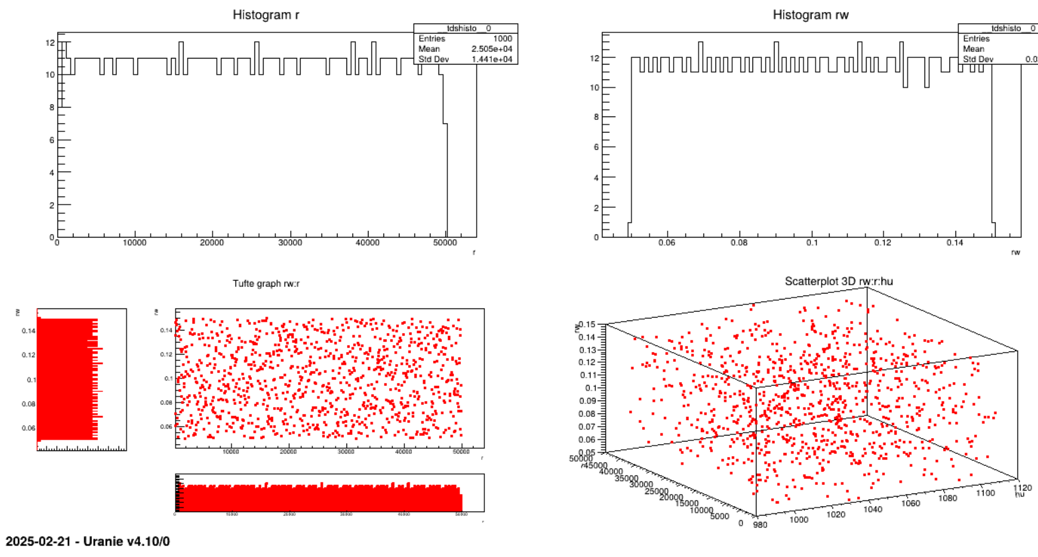


Figure XIV.8: Graph of the macro "samplingFlowrate.C"

XIV.3.2 Macro "samplingLHS.C"

XIV.3.2.1 Objective

Generate a design-of-experiments of 5000 patterns, using the LHS method, with three random attributes:

1. Attribute *x1* obeys an uniform law on interval [3, 4];
2. Attribute *x2* obeys a normal law with mean value equal to 0.5 and standard deviation set to 1.5;
3. Attribute *x3* follows a triangular law on interval [1, 5] with mode 4.

XIV.3.2.2 Macro Uranie

```
{
 // Create a TDataServer
 TDataServer * tds = new TDataServer();
 // Fill the DataServer with the three attributes of the study
 tds->addAttribute(new TUniformDistribution("x1", 3., 4.));
 tds->addAttribute(new TNormalDistribution("x2", 0.5, 1.5));
 tds->addAttribute(new TTriangularDistribution("x3", 1., 5., 4.));

 // Generate the sampling from the TDataServer
 TSampling *sampling = new TSampling(tds, "lhs", 5000);
 sampling->generateSample();

 tds->StartViewer();

 // Graph
 TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro sampling",5,64,1270,667);
 TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
 pad->Divide(2,2);
```

```
  pad->cd(1); tds->Draw("x1");
  pad->cd(2); tds->Draw("x2");
  pad->cd(3); tds->Draw("x3");
  pad->cd(4); tds->drawTufte("x1:x2");

}
```

Laws are set for each attribute and linked to a `TDataServer`:

```
tds->addAttribute(new TUniformDistribution("x1", 3., 4.));
tds->addAttribute(new TNormalDistribution("x2", 0.5, 1.5));
tds->addAttribute(new TTriangularDistribution("x3", 1., 5., 4.));
```

The sampling is generated with the LHS method!

```
TSampling *sampling = new TSampling(tds, "lhs", 5000);
sampling->generateSample();
```

XIV.3.2.3 Graph

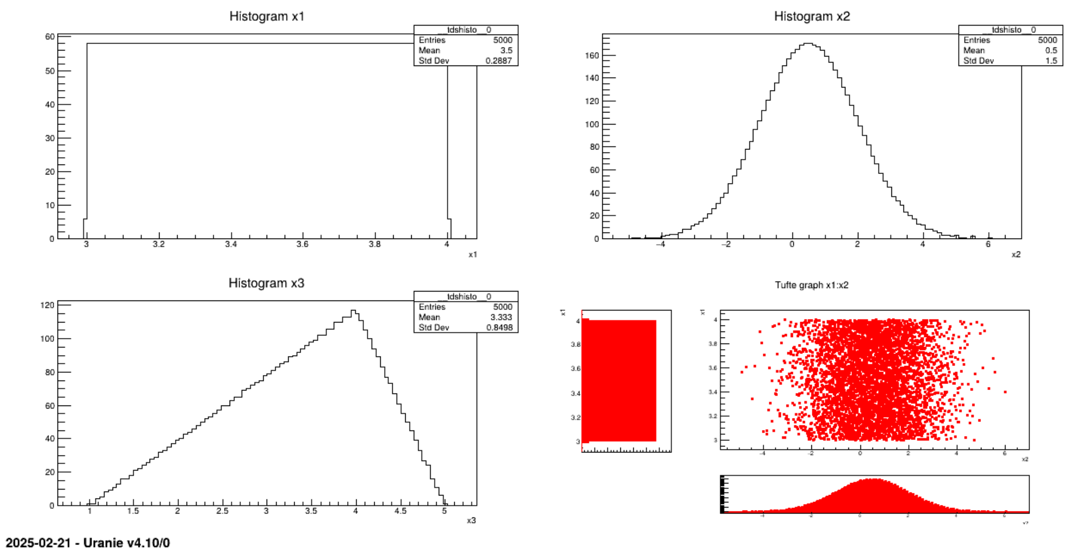


Figure XIV.9: Graph of the macro "samplingLHS.C"

XIV.3.3 Macro "samplingLHSCorrelation.C"

XIV.3.3.1 Objective

Generate a design-of-experiments, of 3000 patterns using the LHS method, with 3 random attributes and taking into account a correlation between the first two attributes. This correlation, equals to 0.99, is computed using the rank values (the *Spearmann* definition, *c.f.* Section III.3 and in [**?**] for more details).

1. Attribute *x1* follows an uniform law on interval [3, 4];
2. Attribute *x2* follows a normal law with a mean value equal to 0.5 and 1.5 as standard deviation;
3. Attribute *x3* obeys a triangular law on interval [1, 5] with mode 4.

XIV.3.3.2 Macro Uranie

```
{
  // Create a TDataServer
  TDataServer * tds = new TDataServer();
  // Fill the DataServer with the three attributes of the study
  tds->addAttribute(new TUniformDistribution("x1", 3., 4.));
  tds->addAttribute(new TNormalDistribution("x2", 0.5, 1.5));
  tds->addAttribute(new TTriangularDistribution("x3", 1., 5., 4.));

  // Generate the sampling from the TDataServer
  TSampling *sampling = new TSampling(tds, "lhs", 3000);
  sampling->setUserCorrelation(0, 1, 0.99);
  sampling->generateSample();

  tds->exportData("toto.dat","x1:x3");
  tds->exportDataHeader("toto.h","x1:x2");

  // Graph
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro samplingCorrelation" ↩
      ,5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2,2);

  pad->cd(1); tds->Draw("x1");
  pad->cd(2); tds->Draw("x2");
  pad->cd(3); tds->Draw("x3");
  pad->cd(4); tds->drawTufte("x1:x2");

}
```

Laws are set for each attribute and linked to a `TDataServer`:

```
tds->addAttribute(new TUniformDistribution("x1", 3., 4.));
tds->addAttribute(new TNormalDistribution("x2", 0.5, 1.5));
tds->addAttribute(new TTriangularDistribution("x3", 1., 5., 4.));
```

The sampling is generated with the LHS method and a correlation is set between the two first attributes:

```
TSampling *sampling = new TSampling(tds, "lhs", 3000);
sampling->setUserCorrelation(0, 1, 0.99);
sampling->generateSample();
```

Data are partially exported in an ASCII file and a header file:

```
tds->exportData("toto.dat","x1:x3");
tds->exportDataHeader("toto.h","x1:x2");
```

XIV.3.3.3 Graph

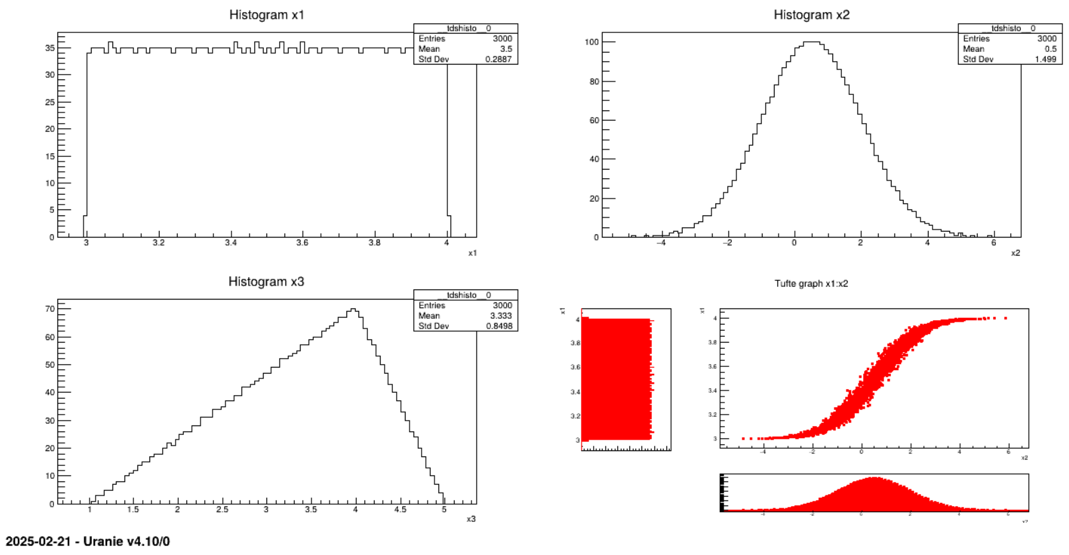


Figure XIV.10: Graph de la macro "samplingLHSCorrelation.C"

XIV.3.4 Macro "samplingQMC.C"

XIV.3.4.1 Objective

Generate a design-of-experiments of 100 patterns using **quasi Monte-Carlo** methods (**"Halton"** or **"Sobol"**) with 12 random attributes, following an uniform law on [3.,2.]. All these information are introduced by variables which will be used by the rest of the macro.

XIV.3.4.2 Macro Uranie

```
{
  // Parameters
  Int_t nSampler = 100;
  TString sQMC ="halton";      // halton / sobol
  Int_t nVar = 12;

  // Create a TDataServer
  TDataServer * tds = new TDataServer();

  // Fill the DataServer with the nVar attributes of the study
  for (Int_t ivar=0; ivar<nVar; ivar++)
    tds->addAttribute( new TUniformDistribution(Form("x%d", ivar+1), -3.0, 2.0));

  // Generate the quasi Monte-Carlo sequence from the TDataServer
  TQMC * qmc = new TQMC(tds, sQMC, nSampler);
  qmc->generateSample();

  // Visualisation
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro qmc",5,64,1270,667);
  Canvas->Range(0,0,25,18);
  TPaveLabel *pl = new TPaveLabel(1,16.3,24,17.5, Form("qMC sequence : %s", sQMC.Data())," ↩
      br");
  pl->SetBorderSize(0);
  pl->Draw();

  pad1 = new TPad("pad1", "Determ", 0.02,0.05,0.48,0.88);
  pad2 = new TPad("pad2","Stoch",0.52,0.05,0.98,0.88);
  pad1->Draw();
  pad2->Draw();

  pad1->cd(); tds->drawTufte("x2:x1");
  pad2->cd(); tds->drawTufte("x11:x12");

}
```

Laws are set for each attribute and linked to a `TDataServer`:

```
for (Int_t ivar=0; ivar<nVar; ivar++)
tds->addAttribute( new TUniformDistribution(Form("x%d", ivar+1), -3.0, 2.0));
```

The sampling is generated with the QMC method and a correlation is set between the two first attributes:

```
TString sQMC ="halton";
TQMC * qmc = new TQMC(tds, sQMC, nSampler);
qmc->generateSample();
```

The rest of the code is used to produce a plot.

XIV.3.4.3 Graph

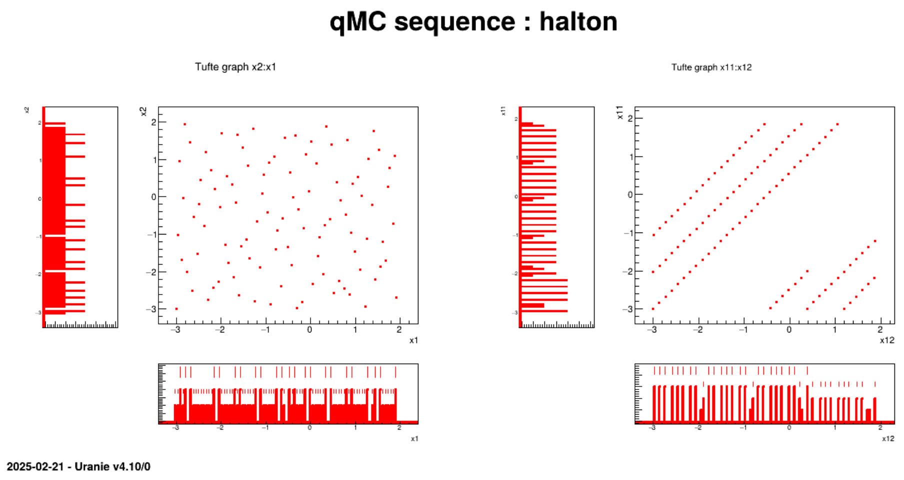


Figure XIV.11: Graph of the macro "`samplingQMC.C`"

XIV.3.5 Macro "`samplingBasicSampling.C`"

XIV.3.5.1 Objective

The objective is to perform three basic samplings, the first with 10000 variables on 10 patterns using a SRS method, the second with 10000 variables on 10 patterns with a LHS method and the third with 10000 variables on 5000 patterns with a SRS method. Each sampling is related to a specific function, and these three functions are run in a fourth one.

XIV.3.5.2 Macro Uranie

```
void test_big_sampling(Bool_t bsave = kFALSE)
{
    TDataServer *tds = new TDataServer();

    cout << "    - Creating attributes..." << endl;
    for(Int_t i = 0; i < 4000; i++)
        tds->addAttribute(new TUniformDistribution(Form("x_%d",i), 0.0, 1.0));

    TBasicSampling *s = new TBasicSampling(tds, "srs", 5000);

    cout << "    - Starting sampling..." << endl;
    s->generateSample();

    if ( bsave ) {
      // WARNING: The generated file is quite big (~300 Mo)
      cout << "    - Saving data..." << endl;
      tds->exportData("_sampling_basic_sampling_big_sampling_.dat");
    } else {
      cout << "    - No saving data (bsave = kFALSE); and the generated file is quite big ↩
          (~300 Mo)." << endl;
    }
}

void test_small_srs_sampling(Bool_t bsave = kFALSE)
{
    TDataServer *tds = new TDataServer();

    cout << "    - Creating attributes..." << endl;
    for(Int_t i = 0; i < 10000; i++)
        tds->addAttribute(new TUniformDistribution(Form("x_%d",i), 0.0, 1.0));

    TBasicSampling *s = new TBasicSampling(tds, "srs", 10);

    cout << "    - Starting sampling..." << endl;
    s->generateSample();

    if ( bsave ) {
      cout << "    - Saving data..." << endl;
      tds->exportData("_sampling_basic_sampling_small_srs_sampling_.dat");
    } else
      cout << "    - No saving data (bsave = kFALSE)." << endl;

}

void test_small_lhs_sampling(Bool_t bsave = kFALSE)
{
    TDataServer *tds = new TDataServer();

    cout << "    - Creating attributes..." << endl;
    for(Int_t i = 0; i < 10000; i++)
        tds->addAttribute(new TUniformDistribution(Form("x_%d",i), 0.0, 1.0));

    TBasicSampling *s = new TBasicSampling(tds, "lhs", 10);

    cout << "    - Starting sampling..." << endl;
    s->generateSample();

    if ( bsave ) {
      cout << "    - Saving data..." << endl;
      tds->exportData("_sampling_basic_sampling_small_lhs_sampling_.dat");
    } else
      cout << "     - No saving data (bsave = kFALSE)." << endl;

}

void samplingBasicSampling()
{
  cout << endl << "****************************************************" << endl;
  cout << " ** Test small SRS sampling (10000 attributes, 10 data)" << endl;
  test_small_srs_sampling(kTRUE);
  cout << "****************************************************" << endl;
  gROOT->ls();



  cout << endl << "****************************************************" << endl;
  cout << " ** Test small LHS sampling (10000 attributes, 10 data)" << endl;
  test_small_lhs_sampling(kTRUE);
  cout << "****************************************************" << endl;
  gROOT->ls();



  cout << endl << "****************************************************" << endl;
  cout << " ** Test big sampling (4000 attributes, 5000 data)" << endl;
  test_big_sampling();
  cout << "****************************************************" << endl;
  gROOT->ls();


}
```

XIV.3.6 Macro "samplingOATRegular.C"

XIV.3.6.1 Objective

This part shows the complete code used to produce the console display in Section III.6.3.3.

XIV.3.6.2 Macro Uranie

```
{
   // step 1
   TDataServer *tds = new TDataServer("tdsoat","Data server for simple OAT design");
   tds->addAttribute(new TAttribute("x1"));
   tds->addAttribute(new TAttribute("x2"));

   // step 2
   tds->getAttribute("x1")->setDefaultValue(0.0);
   tds->getAttribute("x2")->setDefaultValue(10.0);

   // step 3
   TOATDesign *oatSampler = new TOATDesign(tds, "regular", 4);

   // step 4
   Bool_t use_percentage = kTRUE;
   oatSampler->setRange("x1", 2.0);
   oatSampler->setRange("x2", 40.0, use_percentage);
```

```
  // step 5
  oatSampler->generateSample();

  // display
  tds->scan();
}
```

XIV.3.7 Macro "samplingOATRandom.C"

XIV.3.7.1 Objective

This part shows the complete code used to produce the console display in Section III.6.3.4.

XIV.3.7.2 Macro Uranie

```
{
  // step 1
  TDataServer *tds = new TDataServer("tdsoat","Data server for simple OAT design");
  tds->addAttribute(new TUniformDistribution("x1", -5.0, 5.0));
  tds->addAttribute(new TNormalDistribution("x2", 11.0, 1.0));

  // step 2
  tds->getAttribute("x1")->setDefaultValue(0.0);
  tds->getAttribute("x2")->setDefaultValue(10.0);

  // step 3
  TOATDesign *oatSampler = new TOATDesign(tds, "lhs", 1000);

  // step 4
  Bool_t use_percentage = kTRUE;
  oatSampler->setRange("x1", 2.0);
  oatSampler->setRange("x2", 40.0, use_percentage);

  // step 5
  oatSampler->generateSample();

  TCanvas c("can","can",10,32,1200,600);
  TPad *apad = new TPad("apad","apad",0, 0.03, 1, 1);
  apad->Draw();
  apad->Divide(2,1);
  apad->cd(1);
  tds->draw("x1","__modified_att__ == 1");
  apad->cd(2);
  tds->draw("x2","__modified_att__ == 2");
}
```

XIV.3.8 Macro "samplingOATMulti.C"

XIV.3.8.1 Objective

This part shows the complete code used to produce the console display in Section III.6.3.5.

XIV.3.8.2 Macro Uranie

```
{
   // step 1
   TDataServer *tds = new TDataServer("tdsoat","Data server for simple OAT design");
   tds->fileDataRead("myNominalValues.dat");

   // step 3
   TOATDesign *oatSampler = new TOATDesign(tds);

   // step 4
   Bool_t use_percentage = kTRUE;
   oatSampler->setRange("x1", 2.0);
   oatSampler->setRange("x2", 40.0, use_percentage);

   // step 5
   oatSampler->generateSample();

   // display
   tds->scan();
  }
```

XIV.3.9 Macro "samplingOATRange.C"

XIV.3.9.1 Objective

This part shows the complete code used to produce the console display in Section III.6.3.6.

XIV.3.9.2 Macro Uranie

```
{
   // step 1
   TDataServer *tds = new TDataServer("tds","Data server for simple OAT design");
   tds->fileDataRead("myNominalValues.dat");

   // step 3
   TOATDesign *oatSampler = new TOATDesign(tds);

   // step 4
   Bool_t use_percentage = kTRUE;
   oatSampler->setRange("x1", "rx1");
   oatSampler->setRange("x2", 40.0, use_percentage);

   // step 5
   oatSampler->generateSample();

   // display
   tds->scan();
  }
```

XIV.3.10 Macro "samplingSpaceFilling.C"

XIV.3.10.1 Objective

This macro shows the usage of the `TSpaceFilling` class and the resulting design-of-experiments in three simple dataserver cases:

- with two uniform distributions
- with one uniform and one gaussian distributions
- with two gaussian distributions

For each of these configurations (represented in the following plot by a line), the three available algorithms are also tested. They are called:

- SaltelliA
- SaltelliB
- Cukier

This kind of design-of-experiments is not intented to be used regurlarly, it is requested only by few mechanisms like the FAST and RBD methods which rely on fourier transformations. This macro and, above all, the following plot, is made mainly for illustration purpose.

XIV.3.10.2 Macro Uranie

```
void GenerateAndDrawIt(TPad* pad, TDataServer *tds, TSpaceFilling *tsp, int nb, const char ↩
    *title)
{
    pad->cd(nb+1);
    tds->deleteTuple();
    tsp->generateSample();
    tds->drawTufte("x2:x1");
    ((TPaveLabel*)(pad->GetPad(nb+1)->GetPrimitive("TPave")))->SetLabel("");
    ((TH1F*)gPad->GetPrimitive("htemp"))->SetTitle(title);
    gPad->Modified();
}

void samplingSpaceFilling()
{
    //Attributes
    TUniformDistribution *att1 =  new TUniformDistribution("x1",10,12);
    TUniformDistribution *att2 =  new TUniformDistribution("x2",0,3);
    TNormalDistribution *nor1 =  new TNormalDistribution("x1",0,1);
    TNormalDistribution *nor2 =  new TNormalDistribution("x2",3,5);

    // Pointer to DataServer and Samplers
    TDataServer *tds[3];
    TSpaceFilling *tsp[9];
    string algoname[3]={"SaltelliA","SaltelliB","Cukier"};

    // Canvas to produce the 3x3 plot
    TCanvas *Can = new TCanvas("Can","Can",10,32,1200,1200);
    TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
    pad->Divide(3,3);
    int counter=0;

    for(unsigned int itds=0; itds<3; itds++)
    {
  // Create a DataServer to store new configuration (UnivsUni, GausvsUni, Gaus vs Gaus)
  tds[itds] = new TDataServer("test","test");
  switch(itds)
  {
```

```
  case 0: tds[itds]->addAttribute(att1);  tds[itds]->addAttribute(att2); break;
  case 1: tds[itds]->addAttribute(att1);  tds[itds]->addAttribute(nor2); break;
  case 2: tds[itds]->addAttribute(nor1);  tds[itds]->addAttribute(nor2); break;
  }

  // Looping over the 3 spacefilling algo available
  for(unsigned int ialg=0; ialg<3; ialg++)
  {
      // Instantiate the sampler
      switch(ialg)
      {
      case 0: tsp[counter] = new TSpaceFilling(tds[itds], "srs", 1000, TSpaceFilling:: ↩
          kSaltelliA); break;
      case 1: tsp[counter] = new TSpaceFilling(tds[itds], "srs", 1000, TSpaceFilling:: ↩
          kSaltelliB); break;
      case 2: tsp[counter] = new TSpaceFilling(tds[itds], "srs", 1000, TSpaceFilling:: ↩
          kCukier);  break;
      }
      //Draw with correct legend
      GenerateAndDrawIt(pad, tds[itds], tsp[counter], counter, algoname[ialg].c_str());
            counter++;
  }
    }}
```

XIV.3.10.3 Graph

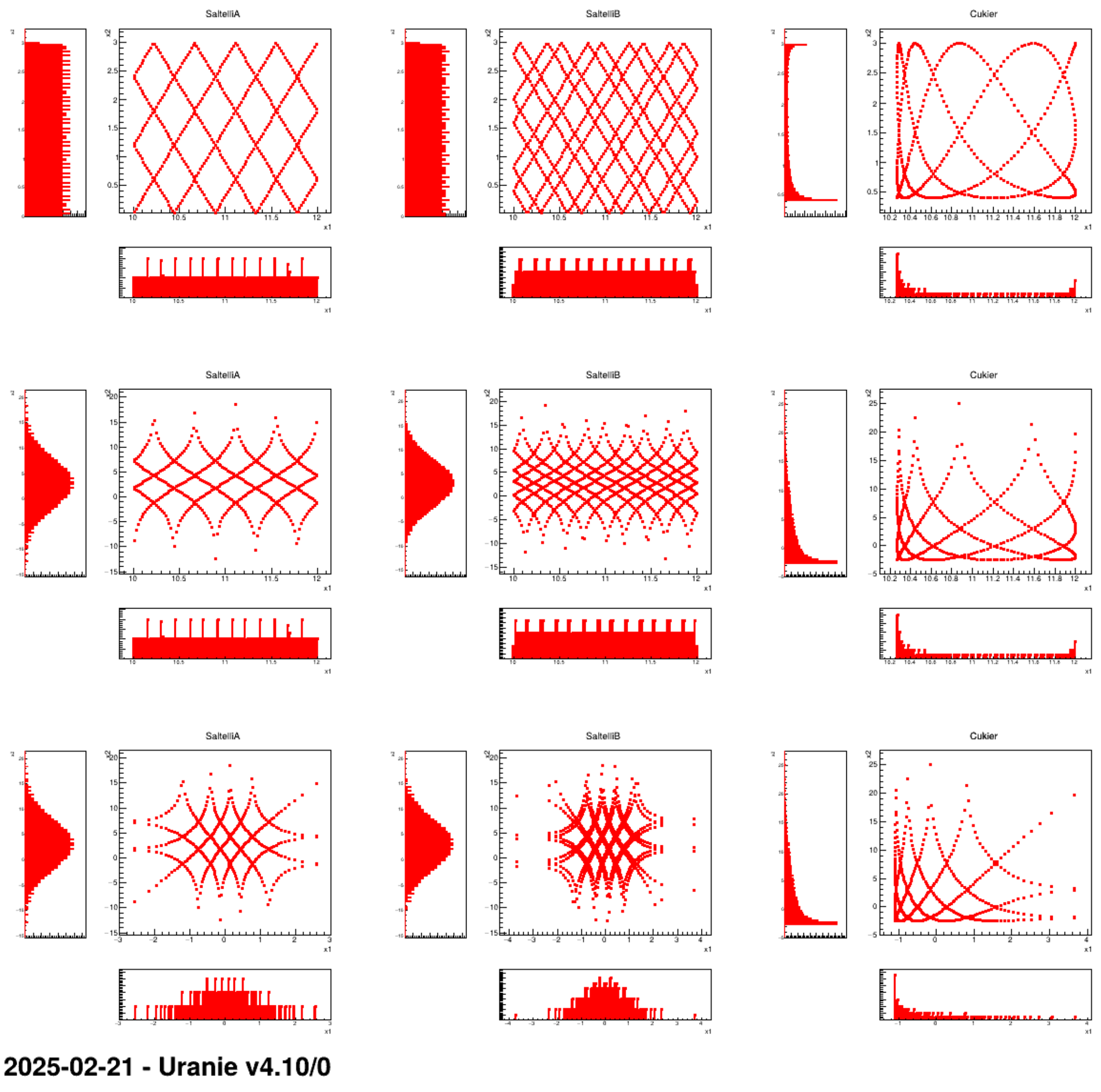


Figure XIV.12: Graph of the macro "`samplingSpaceFilling.C`"

XIV.3.11 Macro "`samplingMaxiMinLHSFromLHSGrid.C`"

XIV.3.11.1 Objective

This macro shows the usage of the `TMaxiMinLHS` class in the case where it is used with an already provided LHS grid. The class itself can generate a LHS grid from scratch (on which the simulated annealing algorithm will be applied to get a maximin grid) but the idea for this macro is to do this procedure in two steps to be able to compare the original LHS grid and the results of the optimisation. The orginal design-of-experiments is done with two uniformly-distributed variables.

The resulting design-of-experiments presented is presented side-by-side with the original one and the mindist criterion calculated is displayed on top of both grid, for illustration purpose.

XIV.3.11.2 Macro Uranie

```
void niceplot(TDataServer *tds, TLatex *lat, string Title)
{

    tds->getTuple()->SetMarkerStyle(20); tds->getTuple()->SetMarkerSize(1);
    tds->Draw("X2:X1");
    stringstream sstr; sstr.str("");
    sstr<<Title<<", MinDist="<<TMaxiMinLHS::getMinDist(tds->getMatrix());
    ((TH2F*)gPad->GetPrimitive("__tdshisto__0"))->SetTitle( "" );
    ((TH2F*)gPad->GetPrimitive("__tdshisto__0"))->GetXaxis()->SetRangeUser(0.,1.);
    ((TH2F*)gPad->GetPrimitive("__tdshisto__0"))->GetYaxis()->SetRangeUser(0.,1.);
    lat->DrawLatex(0.25,0.94,sstr.str().c_str());

}

void samplingMaxiMinLHSFromLHSGrid()
{

    // Canvas to produce the 2x1 plot to compare LHS designs
    TCanvas *Can = new TCanvas("Can","Can",10,32,1200,550);
    TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1);
    pad->Draw();
    pad->Divide(2,1);

    int size = 20; // Size of the samples to be produced
    TLatex *lat = new TLatex(); lat->SetNDC(); lat->SetTextSize(0.038); // To write titles

    // Create dataserver and define the attributes
    TDataServer *tds = new TDataServer("tds","pouet");
    tds->addAttribute( new TUniformDistribution("X1",0,1) ); tds->getAttribute("X1")-> ↩
        delShare(); // Define attribute and state to tds that it owns it
    tds->addAttribute( new TUniformDistribution("X2",0,1) ); tds->getAttribute("X2")-> ↩
        delShare(); // Define attribute and state to tds that it owns it

    // Generate the original LHS grid
    TSampling *sampl = new TSampling(tds,"lhs",size);
    sampl->generateSample();

    // Display it
    pad->cd(1);
    niceplot(tds, lat, "Original LHS");

    // Transform the grid in a maximin LHD
    // Set initial temperature to 0.1, c factor to 0.99 and loop limitations to 300  ↩
        following official recommandation in methodology manual
    TMaxiMinLHS *maxim = new TMaxiMinLHS(tds, size, 0.1, 0.99, 300, 300);
    maxim->generateSample();

    // Display it
    pad->cd(2);
    niceplot(tds, lat, "MaximMin LHS");


  }
```

The macro very much looks like any other design-of-experiments generating macro above: the dataserver is created and the problem is defined along with the input variables. A LHS grid is generated through the use of `TSampling` and display in the first part of the canvas, calling a generic function `niceplot` defined on top of this macro. The new part comes with the following lines:

```
// Transform the grid in a maximin LHD
// Set initial temperature to 0.1, c factor to 0.99 and loop limitations to 300 following ↩
    official recommandation in methodology manual
TMaxiMinLHS *maxim = new TMaxiMinLHS(tds, size, 0.1, 0.99, 300, 300);
maxim->generateSample();
```

The construction line of a `TMaxiMinLHS` is a bit different from the usual `TSampling` object: on top of a pointer to the dataserver, it requires the size of the grid to be generated and the main characteristic of the simulated annealing method to be used for the optimisation of the mindist criteria. A more complete discussion is done on this subject in [30]

XIV.3.11.3 Graph

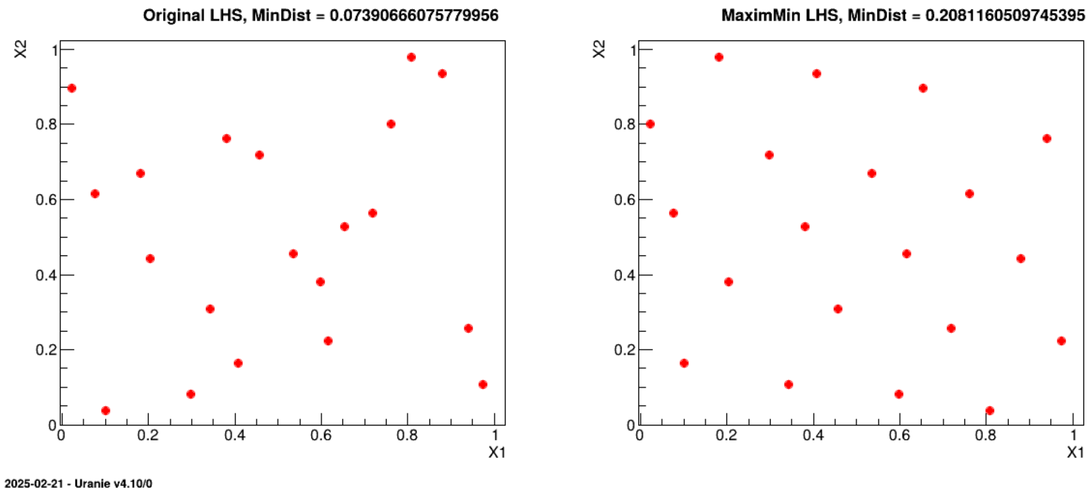


Figure XIV.13: Graph of the macro "`samplingMaxiMinLHSFromLHSGrid.C`"

XIV.3.12 Macro "**`samplingConstrLHSLinear.C`**"

XIV.3.12.1 Objective

This macro shows the usage of the `TConstrLHS` class when one wants to create a constrained LHS with three linear constraints. In order to illustate the concept, it is applied on three input variables drawn from uniform distribution with well-thought boundaries (as the concept of the LHS is to have nicely distributed marginals).

XIV.3.12.2 Macro Uranie

```
#include "ConstrFunctions.C"

void nicegrid(TDataServer *tds, TCanvas *Can)
{
    int ns=tds->getNPatterns(), nx=tds->getNAttributes();
    tds->drawPairs();
    TH1F *h[nx];
    TAttribute *att=NULL;
    gStyle->SetOptStat(0);
    for(int iatt=0; iatt<nx; iatt++)
    {
        Can->GetPad(iatt*(nx+1)+1)->cd();
        att=tds->getAttribute(iatt);
        h[iatt] = new TH1F(Form("%s_histo",att->GetName()), Form("%s_histo;x_%i",att-> ↩
            GetName(), iatt), ns, att->getLowerBound(), att->getUpperBound());
        tds->Draw(Form("%s>>%s_histo", att->GetName(), att->GetName()),"","goff");
        h[iatt]->Draw();
    }
}

int samplingConstrLHSLinear()
{
    // Canvas to produce the 2x1 plot to compare LHS designs
    TCanvas *Can = new TCanvas("Can","Can", 10, 32, 1000, 1000);

    int ns = 250, nx = 3; // Size of the samples to be produced
    // Create dataserver and define the attributes
    TDataServer *tds = new TDataServer("tds","pouet");
    for(int iatt=0; iatt<nx; iatt++)
    {
        tds->addAttribute( new TUniformDistribution(Form("x_%i",iatt),0,iatt+1) );
        tds->getAttribute(Form("x_%i",iatt))->delShare();
    }

    // Generate the constr lhs
    TConstrLHS *constrlhs = new TConstrLHS(tds, ns);
    vector<int> inputs = {1,0,2,1};
    constrlhs->addConstraint(Linear, 2, inputs.size(), &inputs[0]);
    constrlhs->generateSample();

    // Do the plot
    nicegrid(tds, Can);

    return 0;
}
```

The very beginning of these macros is the `nicegrid` method which is here only to show the nice marginal distributions and the scatter plots. One can clearly skip this part to focus on the rest in the main function.

The macro very much looks like any other design-of-experiments generating macro above: the dataserver is created along with the canvas object and the problem is defined along with the input variables and the number of locations to be produced. Once done, then the `TConstrLHS` instance is created with the four following lines:

```
// Generate the constr lhs
TConstrLHS *constrlhs = new TConstrLHS(tds, ns);
vector<int> inputs = {1,0,2,1};
constrlhs->addConstraint(Linear, 2, inputs.size(), &inputs[0]);
constrlhs->generateSample();
```

The constructor is pretty obvious, as it takes only the dataserver object and the number of locations. Once created the main method to be called is the `addConstraint` function which has been largely discussed in Section III.2.4. The first argument of this method is the pointer to the C++ function which has been included in our macro through the very first line:

```
#include "ConstrFunctions.C"
```

which contains the `Linear` function. The rest of the argument are the number of constraints, the size of the list of parameters and its content. Finally the `nicegrid` method is called to produce the nice plot shown in Figure XIV.14

XIV.3.12.3 Graph

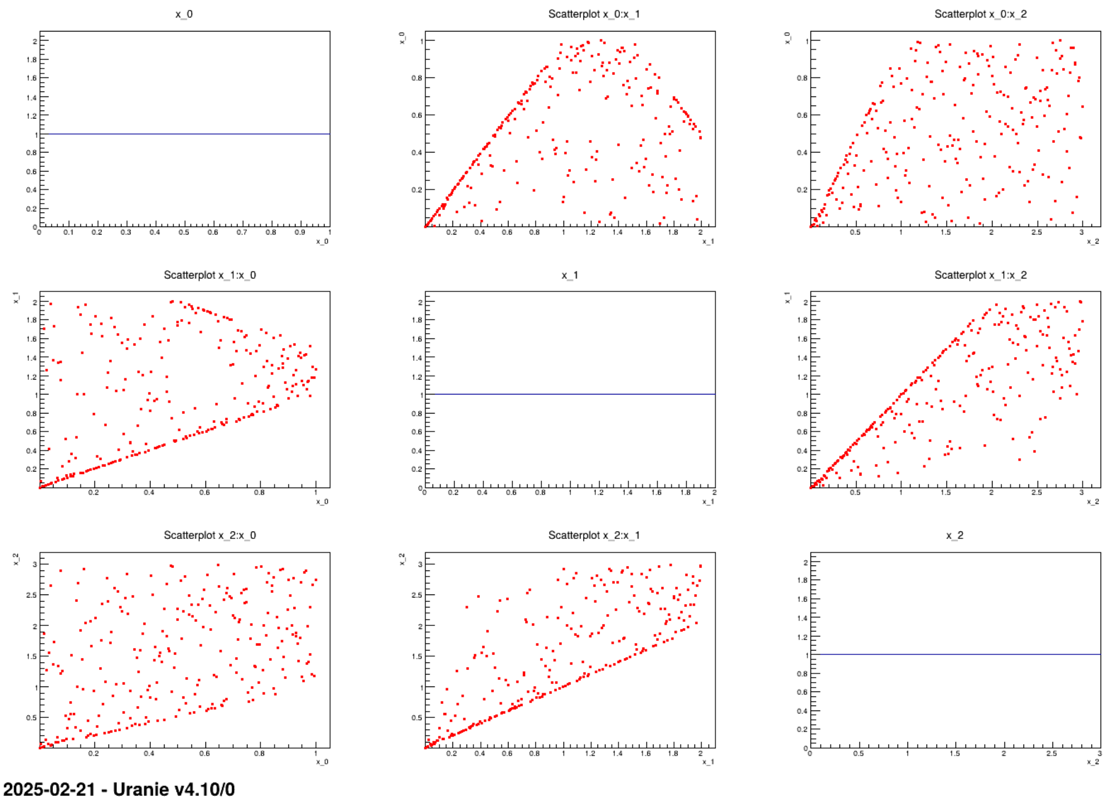


Figure XIV.14: Graph of the macro "samplingConstrLHSLinear.C"

XIV.3.13 Macro "samplingConstrLHSEllipses.C"

XIV.3.13.1 Objective

This macro shows the usage of the TConstrLHS class when one wants to create a constrained LHS with non-linear constraints. In order to illustate the concept, it is applied on three input variables drawn from uniform distribution with well-thought boundaries (as the concept of the LHS is to have nicely distributed marginals). The constraints are excluding the inner part of an ellipse for one of the input plane and the outter part of another ellipse for another input plane.

XIV.3.13.2 Macro Uranie

```
#include "ConstrFunctions.C"

void nicegrid(TDataServer *tds, TCanvas *Can)
{
    int ns=tds->getNPatterns(), nx=tds->getNAttributes();
    tds->drawPairs();
    TH1F *h[nx];
    TAttribute *att=NULL;
    gStyle->SetOptStat(0);
    for(int iatt=0; iatt<nx; iatt++)
    {
        Can->GetPad(iatt*(nx+1)+1)->cd();
        att=tds->getAttribute(iatt);
        h[iatt] = new TH1F(Form("%s_histo",att->GetName()), Form("%s_histo;x_%i",att-> ↩
            GetName(), iatt), ns, att->getLowerBound(), att->getUpperBound());
        tds->Draw(Form("%s>>%s_histo", att->GetName(), att->GetName()),"","goff");
        h[iatt]->Draw();
    }
}

int samplingConstrLHSEllipses()
{
    // Canvas to produce the 2x1 plot to compare LHS designs
    TCanvas *Can = new TCanvas("Can","Can", 10, 32, 1000, 1000);

    int ns = 250, nx = 3; // Size of the samples to be produced
    // Create dataserver and define the attributes
    TDataServer *tds = new TDataServer("tds","pouet");
    for(int iatt=0; iatt<nx; iatt++)
    {
        tds->addAttribute( new TUniformDistribution(Form("x_%i",iatt),0,iatt+1) );
        tds->getAttribute(Form("x_%i",iatt))->delShare();
    }

    // Generate the constr lhs
    TConstrLHS *constrlhs = new TConstrLHS(tds, ns);
    vector<int> inputs = {1,0,1,2};
    constrlhs->addConstraint(CircularRules, 2, inputs.size(), &inputs[0]);
    constrlhs->generateSample();

    // Do the plot
    nicegrid(tds, Can);

    return 0;
}
```

The very beginning of these macros is the nicegrid method which is here only to show the nice marginal distributions and the scatter plots. One can clearly skip this part to focus on the rest in the main function.

The macro very much looks like any other design-of-experiments generating macro above: the dataserver is created along with the canvas object and the problem is defined along with the input variables and the number of locations to be produced. Once done, then the TConstrLHS instance is created with the four following lines:

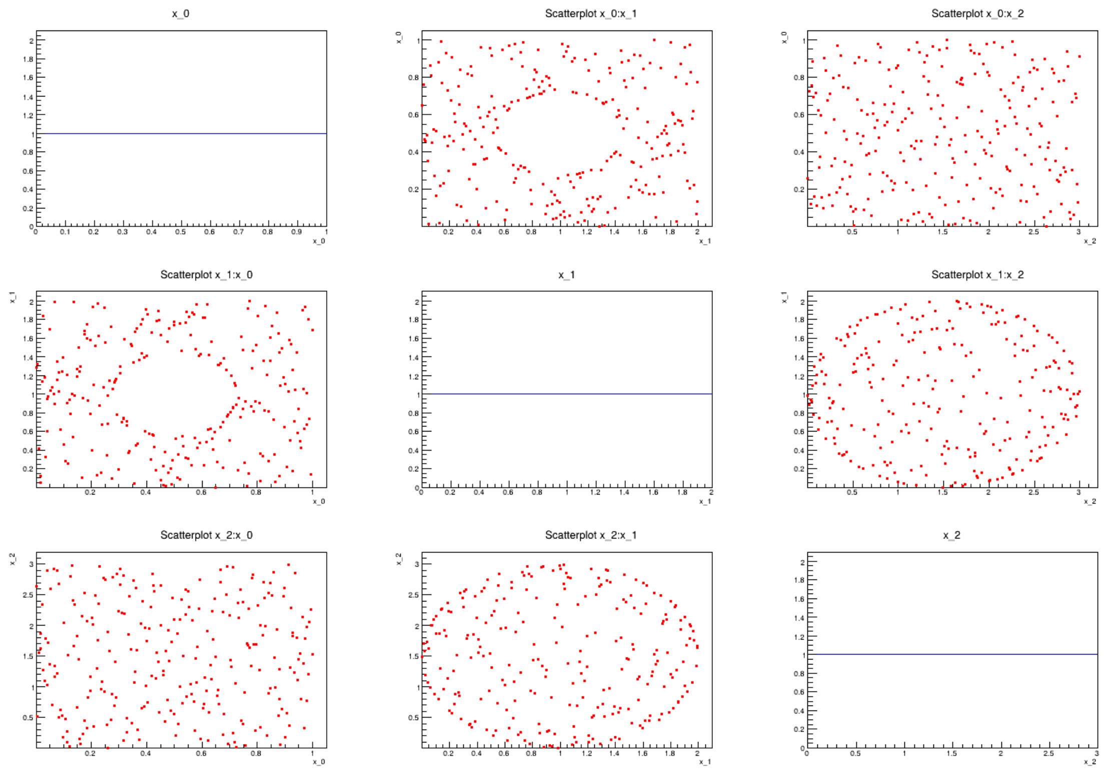
```
// Generate the constr lhs
TConstrLHS *constrlhs = new TConstrLHS(tds, ns);
vector<int> inputs = {1,0,1,2};
constrlhs->addConstraint(CircularRules, 2, inputs.size(), &inputs[0]);
constrlhs->generateSample();
```

The constructor is pretty obvious, as it takes only the dataserver object and the number of locations. Once created the main method to be called is the addConstraint function which has been largely discussed in Section III.2.4. The first argument of this method is the pointer to the C++ function which has been included in our macro through the very first line:

```
#include "ConstrFunctions.C"
```

which contains the CircularRules function. The rest of the argument are the number of constraints, the size of the list of parameters and its content. Finally the nicegrid method is called to produce the nice plot shown in Figure XIV.15

XIV.3.13.3 Graph



2025-02-21 - Uranie v4.10/0

Figure XIV.15: Graph of the macro "samplingConstrLHSEllipses.C"

XIV.3.14 Macro "samplerSingularCorrelationCase.C"

XIV.3.14.1 Objective

This macro shows the usage of the SVD decomposition for the specific case where the target correlation matrix is singular. The idea is to provide a tool that will allow the user to compare quickly both the TSampling and TBasicSampling implementation, with a singular correlation matrix, or not. In order to do that, a toy random correlation matrix generator is provided.

The resulting design-of-experiments is presented side-by-side with the residual of the obtained correlation matrix.

XIV.3.14.2 Macro Uranie

```
{
    // What classes to be used (true==TSampling, false==TBasicSampling)
    bool ImanConover=false;
    TString SamplingType="srs";
    TString SamplerOption="svd"; // Remove svd to use Cholesky

    // Generating randomly a singular correlation matrix
    // dimension of the  problem in total.
    int n = 10;
    // number of dimension self explained.
    // SHOULD BE SMALLER (singular) OR EQUAL TO (full-rank) n.
    int p = 6;
    // number of location in the doe
    int m=300;

    // =========================================================================
    //          Internal recipe to get this correlation matrix
    // =========================================================================
    TMatrixD A(n,p);
    TRandom3 *Rand = new TRandom3();
    for(int i=0;i<n;i++){
        for(int j=0;j<p;j++){
            A(i,j) = Rand->Gaus(0.,1.);
        }
    }

    TMatrixD Gamma(A,TMatrixD::kMultTranspose,A);
    Gamma*=1./n;

    TMatrixD Sig(n,n);
    for(int i=0;i<n;i++) Sig(i,i)=1./sqrt( Gamma(i,i) );

    TMatrixD Corr(Sig,TMatrixD::kMult, Gamma);
    Corr*=Sig;
    // =========================================================================
    //                    Corr is our correlation matrix.
    // =========================================================================

    // Creating the TDS
    TDataServer *tds = new TDataServer("pouet","pouet");

    // Adding attributes
    for(int i=0;i<n;i++) tds->addAttribute( new TNormalDistribution(Form("n%d",i),0.,1.) );

    // Create the sampler and generate the doe
    if(ImanConover){
        TSampling *sam = new TSampling(tds,SamplingType.Data(),m);
        sam->setCorrelationMatrix(Corr);
        sam->generateSample(SamplerOption.Data());
    }else{
        TBasicSampling *sam = new TBasicSampling(tds,SamplingType.Data(),m);
        sam->setCorrelationMatrix(Corr);
        sam->generateCorrSample(SamplerOption.Data());
    }

    // Compute the empirical correlation matrix
    TMatrixD ResultCorr=tds->computeCorrelationMatrix();
    // Change it into a residual matrix
    ResultCorr-=Corr;

    gStyle->SetOptStat(1110);
    // Plot the results
    TCanvas *c=new TCanvas("c","c",1800,900);
    TPad *apad=new TPad("apad","apad",0, 0.03, 1, 1); apad->Draw(); apad->cd();
    apad->Divide(2,1);
    apad->cd(1);
    // Residual distribution
    TH1F *h=new TH1F("h",";#Delta_{#rho}",100,-0.2,0.2);
    for(int i=0;i<n;i++)
    {
        for(int j=i;j<n;j++) h->Fill(ResultCorr(i,j));
    }
    h->Draw();

    // all variables
    apad->cd(2);
    tds->drawPairs();

}
```

This macro starts by a bunch of variables provided to offer many possible configuration, among which:

- use Iman and Conover method or the more simple one to get the correlation (see [30] for a complete description of their respective correlation handling);
- produce a stratified or fully random sample;
- use Cholesky or SVD decomposition to decompose the target correlation matrix;
- use a full-rank or singular correlation matrix. This is allowed thanks to the toy random correlation matrix generator: one has to define the number of variable (**n**) and the number of dimension that should be self-explained (**p**). If the latter is strickly lower than the former, then the correlation matrix generated will be singular, whereas it will be a full-rank one if both quantities are equal.

This is the main part of this macro

```
 // What classes to be used (true==TSampling, false==TBasicSampling)
bool ImanConover=false;
TString SamplingType="srs";
TString SamplerOption="svd"; // Remove svd to use Cholesky

// Generating randomly a singular correlation matrix
// dimension of the  problem in total.
int n = 10;
// number of dimension self explained.
// SHOULD BE SMALLER (singular) OR EQUAL TO (full-rank) n.
int p = 6;
// number of location in the doe
int m=300;
```

Once this is settled, the correlation matrix is created and the dataserver is created and **n** centered-reduced normal attributes are added. The chosen design-of-experiments is generated with the chosen options:

```
// Create the sampler and generate the doe
if(ImanConover){
TSampling *sam = new TSampling(tds,SamplingType.Data(),m);
sam->setCorrelationMatrix(Corr);
sam->generateSample(SamplerOption.Data());
}else{
TBasicSampling *sam = new TBasicSampling(tds,SamplingType.Data(),m);
sam->setCorrelationMatrix(Corr);
sam->generateCorrSample(SamplerOption.Data());
}
```

Finally the obtained design-of-experiments is shown along with the residual of all the correlation coefficients (difference between the target values and the obtained ones). This is shown in Figure XIV.16 and can be used to compare the performance of the samplers.

XIV.3.14.3 Graph

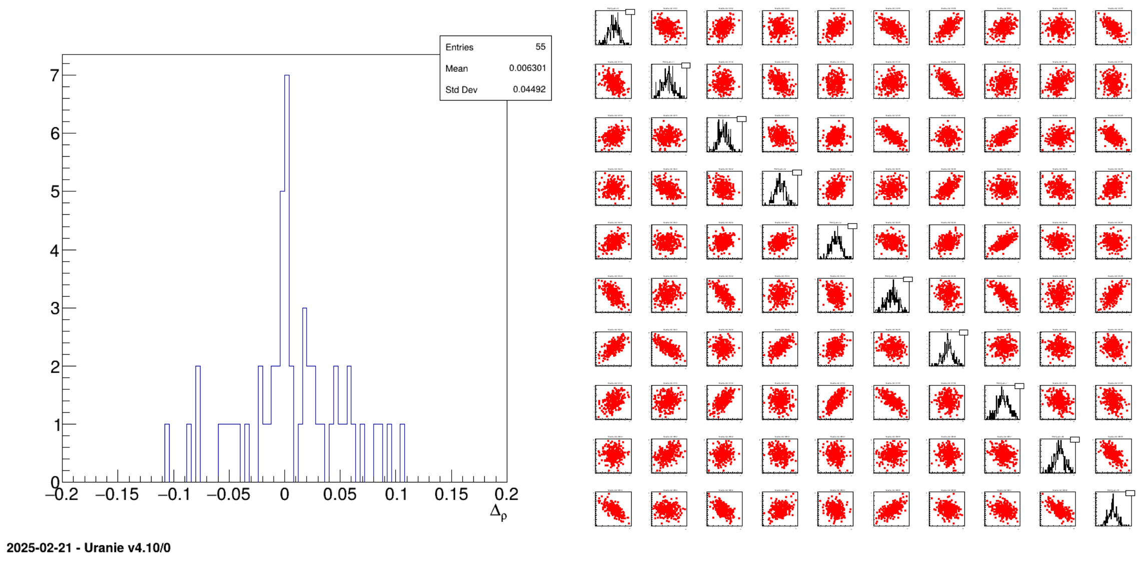


Figure XIV.16: Graph of the macro "`samplerSingularCorrelationCase.C`"

XIV.4 Macros Launcher

XIV.4.1 Macro "`launchFunctionDataBase.C`"

XIV.4.1.1 Objective

Evaluating an analytic function towards a design-of-experiments given in an ASCII file.

This analytic function is written in C++ using the protocol of Uranie in file "`UserFunctions.C`" (the file can be found in the folder *${URANIESYS}/share/uranie/macros/*). This function is named "flowrateModel" and it requires 8 inputs (r_ω, r, T_u, T_l, H_u, H_l, L, K_ω) and returns a scalar (*"ymod"*). The analytic expression of the function is given in (Section IV.1.2.1).

The design-of-experiments is the same as in precedent macro (Section XIV.2.4).

XIV.4.1.2 Macro Uranie

```
{
 // Create a TDataServer
 TDataServer * tds = new TDataServer();
 // Load the data set in the DataServer
 tds->fileDataRead("flowrateUniformDesign.dat");

 // Load the function in the UserFunction macros file
 gROOT->LoadMacro("UserFunctions.C");

 // Create a TLauncherFunction from a TDataServer and an analytical function
 // Rename the outpout attribute "ymod"
 TLauncherFunction * tlf = new TLauncherFunction(tds, "flowrateModel","","ymod");
 // Evaluate the function on all the points in the Design Of Experiments (DoE)
```

```
  tlf->run();

  // Graph
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro loadASCIIFile",5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2,4);
  pad->cd(1); tds->draw("ymod:rw");
  pad->cd(2); tds->draw("ymod:r");
  pad->cd(3); tds->draw("ymod:tu");
  pad->cd(4); tds->draw("ymod:tl");
  pad->cd(5); tds->draw("ymod:hu");
  pad->cd(6); tds->draw("ymod:hl");
  pad->cd(7); tds->draw("ymod:l");
  pad->cd(8); tds->draw("ymod:kw");

}
```

XIV.4.1.3 Graph

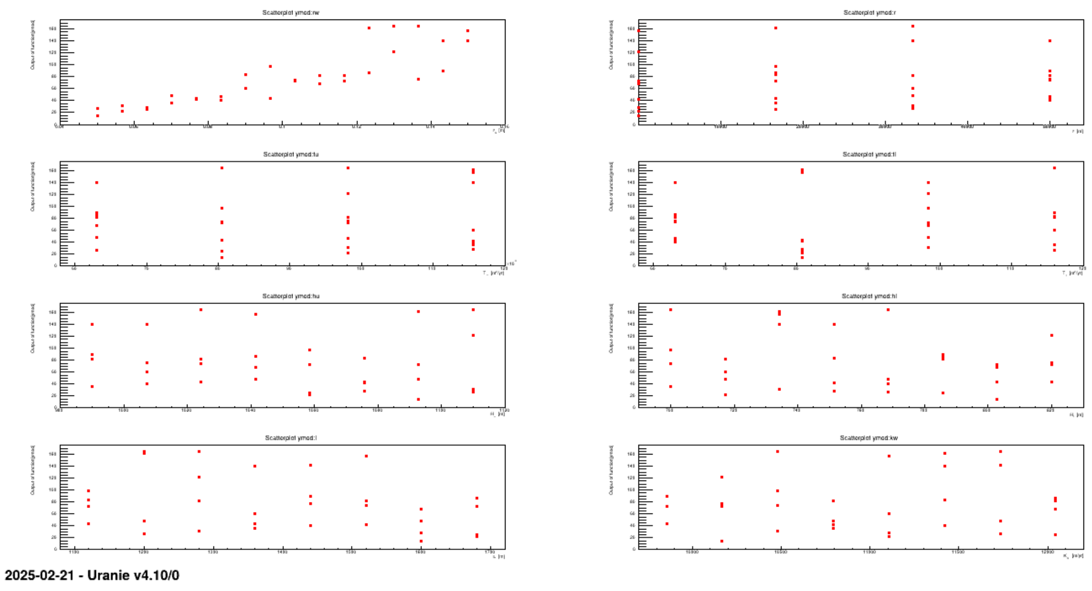


Figure XIV.17: Graph of the macro "launchFunctionDataBase.C"

XIV.4.2 Macro "launchFunctionSampling.C"

XIV.4.2.1 Objective

Evaluating an analytic function in a stochastic design-of-experiments made with the LHS method.

This analytic function is written in C++ using the protocol of Uranie in file "UserFunctions.C" (the file can be found in the folder *${URANIESYS}/share/uranie/macros/*). This function is named "flowrateModel" and it requires 8 inputs (r_ω, r, T_u, T_l, H_u, H_l, L, K_ω) and returns a scalar (*"ymod"*). The analytic expression of the function is given in (Section IV.1.2.1).

The design-of-experiments is made using the LHS method requesting 1000 samples.

XIV.4.2.2 Macro Uranie

```
{
 Int_t nS = 1000;
 // Create a TDataServer
 TDataServer * tds = new TDataServer("tdsFlowrate","TDS for flowrate");
 // Add the eight attributes of the study with uniform law
 tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
 tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
 tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
 tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
 tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
 tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
 tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
 tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

 // Generate the sampling from the TDataServer
 TSampling *sampling = new TSampling(tds, "lhs", nS);
 sampling->generateSample();

 // Load the function in the UserFunction macros file
 gROOT->LoadMacro("UserFunctions.C");

 // Create a TLauncherFunction from a TDataServer and an analytical function
 // Rename the outpout attribute "ymod"
 TLauncherFunction * tlf = new TLauncherFunction(tds, "flowrateModel","","ymod");
 // Evaluate the function on all the design of experiments
 tlf->run();

 // Graph
 TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro loadASCIIFile",5,64,1270,667);
 TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
 pad->Divide(2,4);
 pad->cd(1); tds->drawProfile("ymod:rw","","same");
 pad->cd(2); tds->drawProfile("ymod:r","","same");
 pad->cd(3); tds->drawProfile("ymod:tu","","same");
 pad->cd(4); tds->drawProfile("ymod:tl","","same");
 pad->cd(5); tds->drawProfile("ymod:hu","","same");
 pad->cd(6); tds->drawProfile("ymod:hl","","same");
 pad->cd(7); tds->drawProfile("ymod:l","","same");
 pad->cd(8); tds->drawProfile("ymod:kw","","same");
   }
```

XIV.4.2.3 Graph

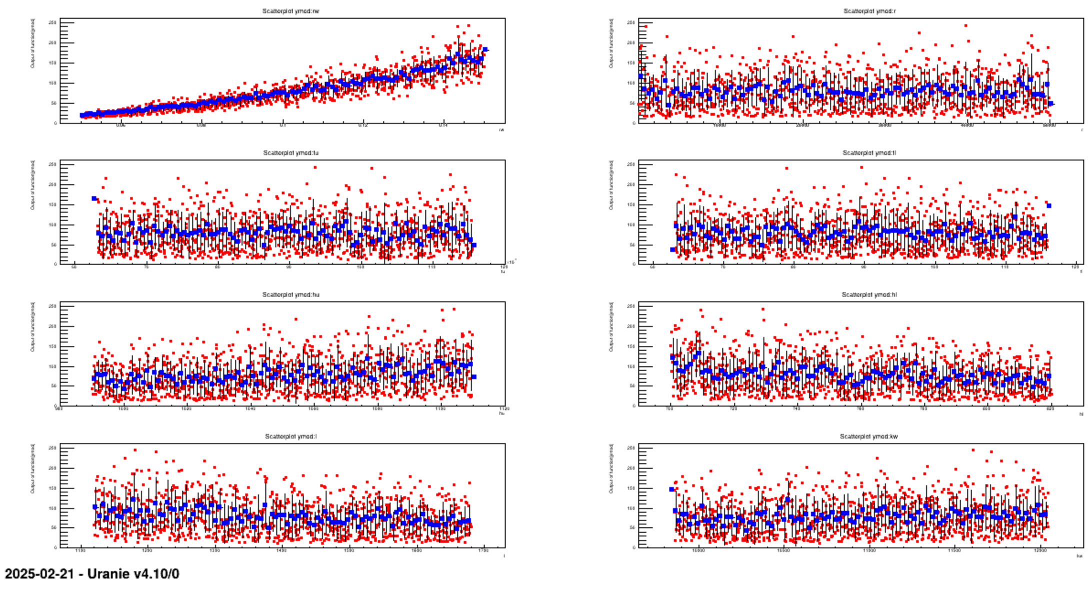


Figure XIV.18: Graph of the macro "`launchFunctionSampling.C`"

XIV.4.3 Macro "`launchFunctionSamplingGraphs.C`"

XIV.4.3.1 Objective

The objective of this macro is to evaluate the `flowrateModel` function on a design-of-experiments, then to perform visualisations with different options. The sampling is made out of 1000 patterns using the LHS method. The `flowrateModel` function is used as a function and uses parameters obeying uniform laws on specific intervals defined in Section IV.1.2.1.

XIV.4.3.2 Macro Uranie

```
{
  Int_t nS = 1000;
  // Create a TDataServer
  TDataServer * tds = new TDataServer("tdsFlowrate","TDS for flowrate");
  // Add the eight attributes of the study with uniform law
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  // Generate the sampling from the TDataServer
  TSampling *sampling = new TSampling(tds, "lhs", nS);
  sampling->generateSample();

  // Load the function in the UserFunction macros file
  gROOT->LoadMacro("UserFunctions.C");

  // Create a TLauncherFunction from a TDataServer and an analytical function
  // Rename the outpout attribute "ymod"
  TLauncherFunction * tlf = new TLauncherFunction(tds, "flowrateModel","","ymod");
  // Evaluate the function on all the design of experiments
  tlf->run();

  // Graph
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro launchFunctionSampling" ↩
      ,5,90,935,614);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2,4);
  pad->cd(1); tds->drawTufte("rw:r","","same");
  pad->cd(2); tds->drawCDF("rw");
  pad->cd(3); tds->draw("ymod");
  pad->cd(4); tds->drawCDF("ymod","","ccdf");
  pad->cd(5); tds->draw("ymod:rw:hu");
  pad->cd(6); tds->drawProfile("ymod:r","","same");
  pad->cd(7); tds->drawProfile("ymod:rw","","same");
  pad->cd(8); tds->draw("ymod:rw","","colz");
    }
```

The attributes linked to the `TDataServer` obey uniform laws:

```
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

The sampling is built with a LHS method on 1000 patterns:

```
TSampling *sampling = new TSampling(tds, "lhs", 1000);
sampling->generateSample();
```

The `flowrateModel` function is evaluated on the design-of-experiments:

```
TLauncherFunction * tlf = new TLauncherFunction(tds, "flowrateModel","","ymod");
tlf->run();
```

Different visualisations are then performed on attributes:

```
tds->drawTufte("rw:r","","same");
tds->drawCDF("rw");
tds->draw("ymod");
tds->drawCDF("ymod","","ccdf");
tds->draw("ymod:rw:hu");
tds->drawProfile("ymod:r","","same");
tds->drawProfile("ymod:rw","","same");
tds->draw("ymod:rw","","colz");
```

XIV.4.3.3 Graph

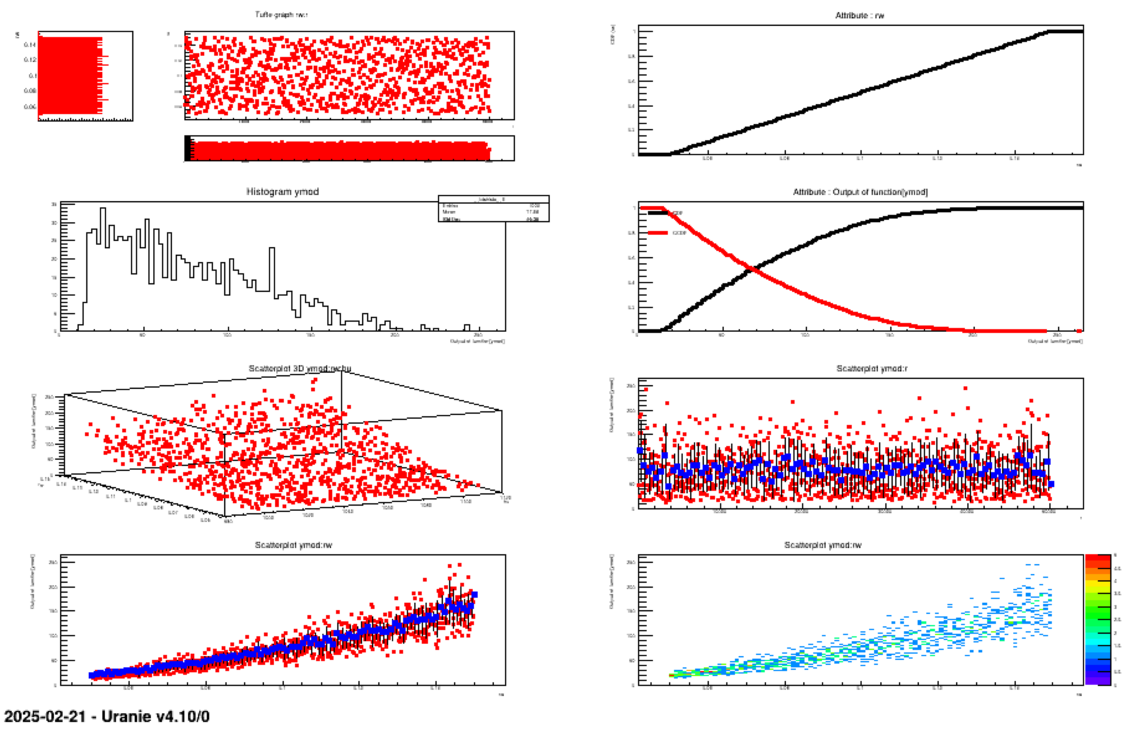


Figure XIV.19: Graph of the macro "launchFunctionSamplingGraphs.C"

XIV.4.4 Macro "launchCodeFlowrateKeyDataBase.C"

XIV.4.4.1 Objective

The objective of the macro is to evaluate the **flowrate** external code on a design-of-experiments where the input file 'flowrate_input_with_keys.in' is a "key=value" type and the output file "_output_flowrate_withRow_.dat" is a "values in rows" type. This **flowrate** code is described in Section IV.1.2.3.1.1.

The database is contained in the file flowrateUniformDesign.dat containing 32 patterns.

```
#NAME: flowrateborehole
#TITLE: Uniform design of flow rate borehole problem proposed by Ho and Xu(2000)
#COLUMN_NAMES: rw| r| tu| tl| hu| hl| l| kw | ystar
#COLUMN_TITLES: r_{#omega}| r | T_{u} | T_{l} | H_{u} | H_{l} | L | K_{#omega} | y^{*}
#COLUMN_UNITS: m | m | m^{2}/yr | m^{2}/yr | m | m | m | m/yr | m^{3}/yr

0.0500 33366.67  63070.0 116.00 1110.00 768.57 1200.0 11732.14  26.18
0.0500   100.00  80580.0  80.73 1092.86 802.86 1600.0 10167.86  14.46
0.0567   100.00  98090.0  80.73 1058.57 717.14 1680.0 11106.43  22.75
0.0567 33366.67  98090.0  98.37 1110.00 734.29 1280.0 10480.71  30.98
0.0633   100.00 115600.0  80.73 1075.71 751.43 1600.0 11106.43  28.33
0.0633 16733.33  80580.0  80.73 1058.57 785.71 1680.0 12045.00  24.60
0.0700 33366.67  63070.0  98.37 1092.86 768.57 1200.0 11732.14  48.65
0.0700 16733.33 115600.0 116.00  990.00 700.00 1360.0 10793.57  35.36
0.0767   100.0  115600.0  80.73 1075.71 751.43 1520.0 10793.57  42.44
0.0767 16733.33  80580.0  80.73 1075.71 802.86 1120.0  9855.00  44.16
0.0833 50000.00  98090.0  63.10 1041.43 717.14 1600.0 10793.57  47.49
0.0833 50000.00 115600.0  63.10 1007.14 768.57 1440.0 11419.29  41.04
```

```
0.0900 16733.33  63070.0 116.00 1075.71 751.43 1120.0 11419.29  83.77
0.0900 33366.67 115600.0 116.00 1007.14 717.14 1360.0 11106.43  60.05
0.0967 50000.00  80580.0  63.10 1024.29 820.00 1360.0  9855.00  43.15
0.0967 16733.33  80580.0  98.37 1058.57 700.00 1120.0 10480.71  97.98
0.1033 50000.00  80580.0  63.10 1024.29 700.00 1520.0 10480.71  74.44
0.1033 16733.33  80580.0  98.37 1058.57 820.00 1120.0 10167.86  72.23
0.1100 50000.00  98090.0  63.10 1024.29 717.14 1520.0 10793.57  82.18
0.1100   100.00  63070.0  98.37 1041.43 802.86 1600.0 12045.00  68.06
0.1167 33366.67  63070.0 116.00  990.00 785.71 1280.0 12045.00  81.63
0.1167   100.00  98090.0  98.37 1092.86 802.86 1680.0  9855.00  72.5
0.1233 16733.33 115600.0  80.73 1092.86 734.29 1200.0 11419.29 161.35
0.1233 16733.33  63070.0  63.10 1041.43 785.71 1680.0 12045.00  86.73
0.1300 33366.67  80580.0 116.00 1110.00 768.57 1280.0 11732.14 164.78
0.1300   100.00  98090.0  98.37 1110.00 820.00 1280.0 10167.86 121.76
0.1367 50000.00  98090.0  63.10 1007.14 820.00 1440.0 10167.86  76.51
0.1367 33366.67  98090.0 116.00 1024.29 700.00 1200.0 10480.71 164.75
0.1433 50000.00  63070.0 116.00  990.00 785.71 1440.0  9855.00  89.54
0.1433 50000.00 115600.0  63.10 1007.14 734.29 1440.0 11732.14 141.09
0.1500 33366.67  63070.0  98.37  990.00 751.43 1360.0 11419.29 139.94
0.1500   100.00 115600.0  80.73 1041.43 734.29 1520.0 11106.43 157.59
```

The input file `flowrate_input_with_keys.in` is "key=value" format:

```
#
#
# INPUT FILE with KEYS for the "FLOWREATE" code
# \date   2008-04-22 12:53:35
#


date = 123456 ;


#########################
##
##  exclude points
##
chu = 1050;
chl = 770;
cr  = 1100;
##
#########################

#########################
##
##  parameters : 8
##
Rw = 0.0500 ;
R = 33366.67 ;
Tu = 63070.0 ;
Tl = 116.00 ;
Hu = 1110.00 ;
Hl = 768.57;
L = 1200.0 ;
Kw = 11732.14 ;
##
#########################


#########################
##
## to simulate CPU time
```

```
##
## normal         1 :
## min     10000000 : 1.160u 0.000s 0:01.16 100.0%
## max    100000000 : 11.600u 0.010s 0:11.61 100.0%
##
nLoop = 1;
##
#########################

end = 6;
```

This file defines the eight variables r_ω, r, T_u, T_l, H_u, H_l, L, K_ω needed to perform the execution of the command **flowrate -k**.

The output file, _output_flowrate_withRow_.dat when created, is a "values in rows with an header" type. It looks like:

```
#COLUMN_NAMES: yhat | d

6.757218e+01  4.092561e+03
```

yhat and **d** have to be defined as output variables in the macro.

XIV.4.4.2 Macro Uranie

```
{
  // Create a TDataServer
  TDataServer * tds = new TDataServer();
  // Load the data base in the DataServer
  tds->fileDataRead("flowrateUniformDesign.dat");

  // The reference input file
  TString sIn = TString("flowrate_input_with_keys.in");

  // Set the reference input file and the key for each input attributes
  tds->getAttribute("rw")->setFileKey(sIn, "Rw");
  tds->getAttribute("r")->setFileKey(sIn, "R");
  tds->getAttribute("tu")->setFileKey(sIn, "Tu");
  tds->getAttribute("tl")->setFileKey(sIn, "Tl");
  tds->getAttribute("hu")->setFileKey(sIn, "Hu");
  tds->getAttribute("hl")->setFileKey(sIn, "Hl");
  tds->getAttribute("l")->setFileKey(sIn, "L");
  tds->getAttribute("kw")->setFileKey(sIn, "Kw");

  // The output file of the code
  TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
  // The attribute in the output file
  fout->addAttribute(new TAttribute("yhat"));

  // Instanciation de mon code
  TCode *mycode = new TCode(tds, "flowrate -s -k");
  // on ajoute le fichier de sortie du code
  mycode->addOutputFile( fout );

  // Lancement du code
  TLauncher *lanceur = new TLauncher(tds, mycode);
  lanceur->setSave();
  lanceur->setClean();
  //lanceur->setWorkingDirectory(gSystem->Getenv("PWD") + TString("/tmpLanceurUranie/ ↩
      flowrate"));
  lanceur->setVarDraw("yhat:rw","","");
  lanceur->run();


  // Graph
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro launchCodeFlowrateKeyDataBase" ↩
      ,5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2,4);

  pad->cd(1); tds->draw("yhat:rw");
  pad->cd(2); tds->draw("yhat:r");
  pad->cd(3); tds->draw("yhat:tu");
  pad->cd(4); tds->draw("yhat:tl");
  pad->cd(5); tds->draw("yhat:hu");
  pad->cd(6); tds->draw("yhat:hl");
  pad->cd(7); tds->draw("yhat:l");
  pad->cd(8); tds->draw("yhat:kw");

}
```

The database is loaded from a data file `flowrateUniformDesign.dat`:

```
tds->fileDataRead("flowrateUniformDesign.dat");
```

Then, properties are set for each input variable. For example for the first variable, it is created as a `TAttribute` named "rw": this `TAttribute` is then linked to the input file with:

```
tds->getAttribute("rw")->setFileKey(TString("flowrate_input_with_keys.in"), "Rw");
```

which will be understood by Uranie as the fact the **rw** variable has to be read in the input file with "key=value" format with the key **Rw**.

We use the output file with "values in rows" type `_output_flowrate_withRow_.dat` which will be instantiated as a `TOutputFileRow`.

```
TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
```

The variables **yhat** and **d** can be linked to this `TOutputFileRow` output file. But here, only **yhat** will be considered by Uranie as variable of interest.

```
fout->addAttribute("yhat");
```

We set the code as being the **flowrate** execution with *"-k"* option.

```
TCode *mycode = new TCode(tds, "flowrate -k");
```

which indicates that **flowrate** code has to find the input file with "key=value" type.

Then the launcher is initialised with a `TDataServer` and the code is executed:

```
TLauncher *lanceur = new TLauncher(tds, mycode);
```

The launcher will execute the **flowrate -k** command for each of the 32 patterns with the `run` method:

```
lanceur->run();
```

XIV.4.4.3 Graph

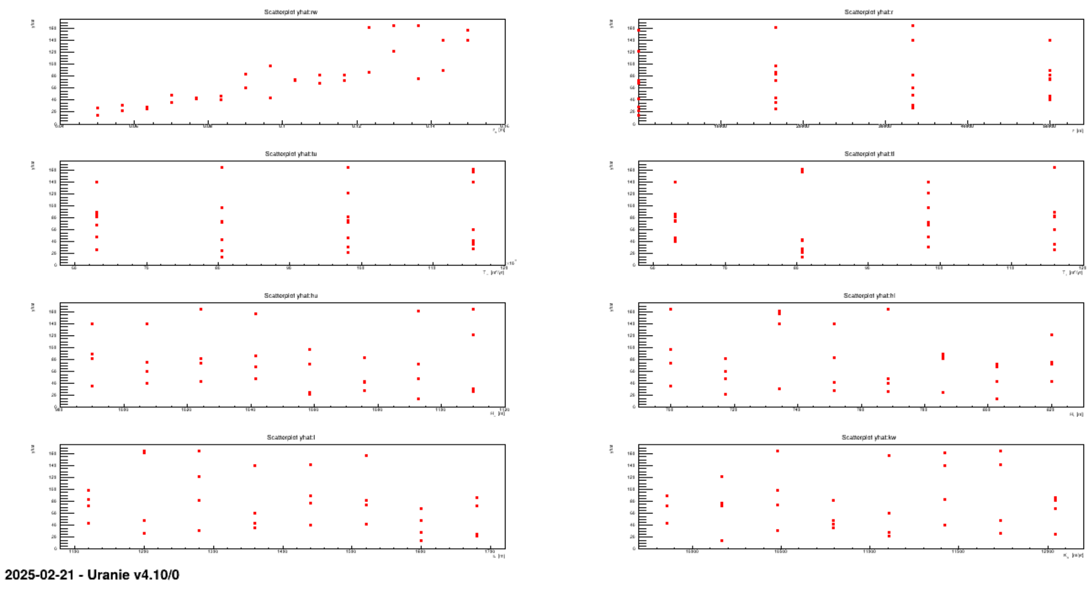


Figure XIV.20: Graph of the macro "`launchCodeFlowrateKeyDataBase.C`"

XIV.4.5 Macro "`launchCodeFlowrateKeySampling.C`"

XIV.4.5.1 Objective

The objective of the macro is to evaluate the **flowrate** external code on a design-of-experiments with the input file "`flowrate_input_with_keys.in`" with "key=value" type and the output file "`_output_flowrate_withRow_.dat`" with "values in rows" type. This **flowrate** code is described in Section IV.1.2.3.1.1.

The used input file `flowrate_input_with_keys.in` is with "key=value" type:

```
#
#
# INPUT FILE with KEYS for the "FLOWREATE" code
# \date   2008-04-22 12:53:35
#


date = 123456 ;


#########################
##
##  exclude points
##
chu = 1050;
chl = 770;
cr  = 1100;
##
#########################

#########################
##
##  parameters : 8
```

```
##
Rw = 0.0500 ;
R = 33366.67 ;
Tu = 63070.0 ;
Tl = 116.00 ;
Hu = 1110.00 ;
Hl = 768.57;
L = 1200.0 ;
Kw = 11732.14 ;
##
#########################


#########################
##
## to simulate CPU time
##
## normal          1 :
## min      10000000 : 1.160u 0.000s 0:01.16 100.0%
## max     100000000 : 11.600u 0.010s 0:11.61 100.0%
##
nLoop = 1;
##
#########################

end = 6;
```

This file defines the eight variables r_ω, r, T_u, T_l, H_u, H_l, L, K_ω needed to perform the execution of the command **flowrate -k**.

The output file, `_output_flowrate_withRow_.dat` is with "values in rows" type. It looks like:

```
#COLUMN_NAMES: yhat | d

6.757218e+01  4.092561e+03
```

yhat and **d** have to be defined as output variables in the macro.

XIV.4.5.2 Macro Uranie

```
void launchCodeFlowrateKeySampling(Int_t nS = 100)
{
  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsFlowrate", "Design of experiments for Flowrate");

  // Add the study attributes ( min, max and nominal values)
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  // The reference input file
  TString sIn = TString("flowrate_input_with_keys.in");

  // Set the reference input file and the key for each input attributes
  tds->getAttribute("rw")->setFileKey(sIn, "Rw");
  tds->getAttribute("r")->setFileKey(sIn, "R");
  tds->getAttribute("tu")->setFileKey(sIn, "Tu");
  tds->getAttribute("tl")->setFileKey(sIn, "Tl");
  tds->getAttribute("hu")->setFileKey(sIn, "Hu");
  tds->getAttribute("hl")->setFileKey(sIn, "Hl");
  tds->getAttribute("l")->setFileKey(sIn, "L");
  tds->getAttribute("kw")->setFileKey(sIn, "Kw");

  // Generate the Design of Experiments
  TSampling *sampling = new TSampling(tds, "lhs", nS);
  sampling->generateSample();

  // The output file of the code
  TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
  // The attribute in the output file
  fout->addAttribute(new TAttribute("yhat"));

  // Instanciation de mon code
  TCode *mycode = new TCode(tds, "flowrate -s -k");
  // on ajoute le fichier de sortie du code
  mycode->addOutputFile( fout );

  // Lancement du code
  TLauncher *lanceur = new TLauncher(tds, mycode);
  lanceur->setSave();
  lanceur->setClean();
  //lanceur->setWorkingDirectory(gSystem->Getenv("PWD") + TString("/tmpLanceurUranie/ ↩
      flowrate"));
  lanceur->setVarDraw("yhat:rw","","");

  lanceur->run();

  tds->exportData("_flowrate_sampler_launcher_.dat");

  // Visualisation
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro launchCodeFlowrateFlag" ↩
      ,5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  gStyle->SetPalette(1);
  pad->Divide(2, 2);
  pad->cd(1); tds->draw("yhat:rw");

  pad->cd(3); tds->draw("yhat");
  pad->cd(2); tds->draw("yhat:rw","","colz");

  pad->cd(4);
  // The parallel plot
  tds->draw("rw:r:tu:tl:hu:hl:l:kw:yhat","","para");
  TParallelCoord* para = (TParallelCoord*)gPad->GetListOfPrimitives()->FindObject(" ↩
      ParaCoord");

  // The output attribute
  TParallelCoordVar* axis = (TParallelCoordVar*)para->GetVarList()->FindObject("yhat");
  axis->AddRange(new TParallelCoordRange(axis,15.0,80.0));
  para->AddSelection("blue");
  para->GetCurrentSelection()->SetLineColor(kBlue);

}
```

The laws of distribution are set for each input variable. For example for the **Rw** variable, the macro creates an uniform distribution between 0.05 and 0.15 associated to a `TAttribute` named **"rw"**:

```
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
```

Then, the `TAttribute` is linked to the input file with:

```
tds->getAttribute("rw")->setFileKey(TString("flowrate_input_with_keys.in"), "@Rw@");
```

which will be understood by Uranie as the fact the **rw** variable has to be read in the input file with "key=value" format with the key **Rw**.

A design-of-experiments is then built with 100 samplings using the LHS method:

```
TSampling *sampling = new TSampling(tds, "lhs", 100);
sampling->generateSample();
```

We want the output file to be with values in rows. So the output file `_output_flowrate_withRow_.dat` is set as a `TOutputFileRow`:

```
TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
```

The variable **yhat** and **d** could be linked to this `TOutputFileRow` output file. But here, only **yhat** will be considered as variable of interest by Uranie:

```
fout->addAttribute("yhat");
```

We set the code as being the **flowrate** execution with *"-k"* option:

```
TCode *mycode = new TCode(tds, "flowrate -k");
```

in xhich the "-k" option indicates that **flowrate** has to find input files with "key=value" format.

Then the launcher is initialised with a `TDataServer` and the code is executed:

```
TLauncher *lanceur = new TLauncher(tds, mycode);
```

The launcher will execute the **flowrate -k** command for each of the 100 patterns with the `run` method:

```
tlch->run();
```

XIV.4.5.3 Graph

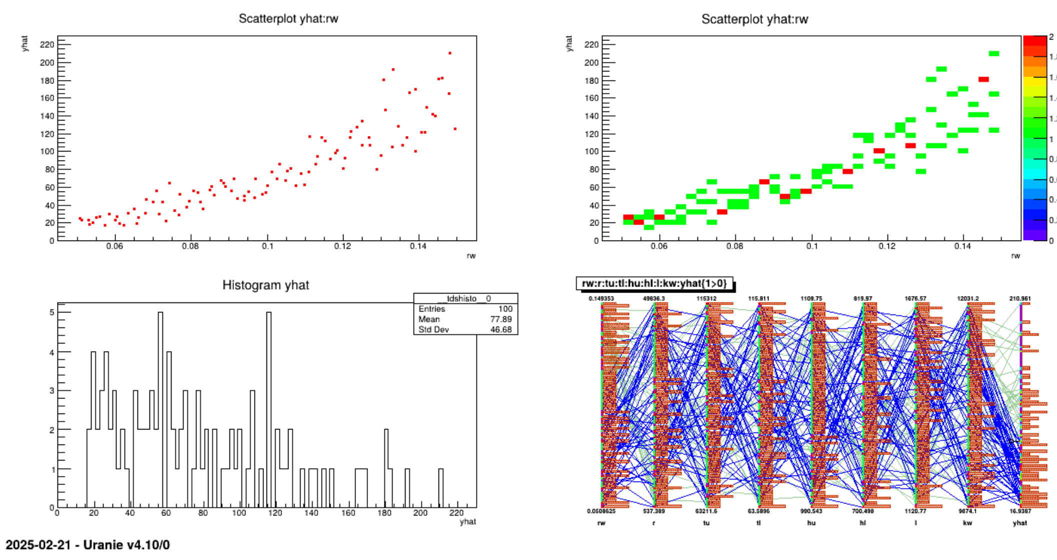


Figure XIV.21: Graph of the macro "launchCodeFlowrateKeySampling.C"

XIV.4.6 Macro "**launchCodeFlowrateXMLSampling.C**"

XIV.4.6.1 Objective

The objective of the macro is to evaluate the **flowrate** external code on a design-of-experiments with the input file "flowrate_input_with_xml.in" with XML format and the output file "_output_flowrate_withXML_.dat" also with XML format. This **flowrate** code is described in Section IV.1.2.3.1.1.

The used input file flowrate_input_with_xml.in is with XML type:

```
<?xml version="1.0"?>
<problem>
  <description name="flowrate" title="UseCase flowrate with XML input file" version="1.0"  ↩
      date="2011-07-22 12:55:17">
    <tool name="uranie" version="0.3"/>
  </description>

  <steady_state name="sch">
    <wall_friction rw="0.0500" r="33366.67"/>
    <tinit>0.0</tinit>
    <tmax>1000000</tmax>
    <dt_step_nb_max>1500</dt_step_nb_max>
    <parameter>
      <tonode>mesher</tonode><toport>dt_hu</toport>
      <value><double>1110.00</double></value>
    </parameter>
    <parameter>
      <tonode>mesher</tonode><toport>dt_hl</toport>
      <value><double>768.57</double></value>
    </parameter>
    <parameter>
      <tonode>mesher</tonode><toport>dt_hd</toport>
      <value><int>12345</int></value>
    </parameter>
    <facsec>1000000.</facsec>
    <kW value="11732.14"/>
    <informations>
      <parameter name="Tu">
        <Uniform_Field>1</Uniform_Field>
        <value><double>63070.0</double></value>
      </parameter>
      <parameter name="Tl">
        <Uniform_Field>1</Uniform_Field>
        <value><double>116.00</double></value>
      </parameter>
      <parameter name="L" precision="1200.0"/>
    </informations>

    <convergence>
      <criterion>relative_max_du_dt</criterion>
      <precision>1.e-6</precision>
    </convergence>

    <stop_criterium ch_abscissa_hu="1050" ch_ordinate_hl="770" c_radius="1100" nLoop="1"/>

    <solver name="Newton3">
      <max_iter_matrix>1</max_iter_matrix>
      <max_iter_implicit>1</max_iter_implicit>
      <date>5654321</date>
      <implicit_convergence_threshold>1.e-6</implicit_convergence_threshold>
      <implicit_assembly>10</implicit_assembly>
      <linear_solver name="BiCGS">
        <preconditioner>ILU</preconditioner>
        <implicit_solve_threshold>1.e-5</implicit_solve_threshold>
      </linear_solver>
    </solver>

  </steady_state>
</problem>
```

This file defines the eight variables r_ω, r, T_u, T_l, H_u, H_l, L, K_ω needed to perform the execution of the command **flowrate -x**.

The output file, `_output_flowrate_withXML_.dat` is with XML type. It looks like:

```
<?xml version="1.0"?>
<steady_state name="flowrate">
  <parameter>
    <tonode>mesher</tonode>
    <toport>dt_hl</toport>
    <value>
      <double>2.618019e+01</double>
    </value>
  </parameter>
  <distance value="3.602045e+03"/>
</steady_state>
```

yhat and **d** have to be defined as output variables in the macro.

XIV.4.6.2 Macro Uranie

```
void launchCodeFlowrateXMLSampling(Int_t nS = 100){
  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsFlowrate", "Design of experiments for Flowrate");
```

```
// Add the study attributes ( min, max and nominal values)
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

// The reference input file
TString sIn = TString("flowrate_input_with_xml.in");

tds->getAttribute("rw")->setFileKey(sIn, "wall_friction/@rw", "%e", TAttributeFileKey:: ↩
    kXMLAttribute);
tds->getAttribute("r")->setFileKey(sIn, "wall_friction/@r", "%e", TAttributeFileKey:: ↩
    kXMLAttribute);
tds->getAttribute("tu")->setFileKey(sIn, "parameter[@name='Tu']/value/double", "%e",  ↩
    TAttributeFileKey::kXMLField);
tds->getAttribute("tl")->setFileKey(sIn, "parameter[@name='Tl']/value/double", "%e",  ↩
    TAttributeFileKey::kXMLField);
tds->getAttribute("hu")->setFileKey(sIn, "parameter[tonode='mesher' and toport='dt_hu']/ ↩
    value/double", "%e", TAttributeFileKey::kXMLField);
tds->getAttribute("hl")->setFileKey(sIn, "parameter[tonode='mesher' and toport='dt_hl']/ ↩
    value/double", "%e", TAttributeFileKey::kXMLField);
tds->getAttribute("l")->setFileKey(sIn, "parameter[@name='L']/@precision", "%e",  ↩
    TAttributeFileKey::kXMLAttribute);
tds->getAttribute("kw")->setFileKey(sIn, "kW/@value", "%e", TAttributeFileKey:: ↩
    kXMLAttribute);


// Generate the Design of Experiments
TSampling *sampling = new TSampling(tds, "lhs", nS);
sampling->generateSample();


// The output file of the code
TOutputFileXML *fout = new TOutputFileXML("_output_flowrate_withXML_.dat");
// The attribute in the output file
fout->addAttribute(new TAttribute("yhat"), "/steady_state[@name='flowrate']/parameter/ ↩
    value/double", TAttributeFileKey::kXMLField);

// Instanciation de mon code
TCode *mycode = new TCode(tds, "flowrate -s -x");
// on ajoute le fichier de sortie du code
mycode->addOutputFile( fout );

// Lancement du code
TLauncher *lanceur = new TLauncher(tds, mycode);
lanceur->setSave();
lanceur->setClean();
lanceur->setVarDraw("yhat:rw","","");

lanceur->run();

tds->exportData("_flowrate_sampler_launcher_.dat");


// Visualisation
TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro launchCodeFlowrateOATMinMax" ↩
    ,5,64,1270,667);
```

```
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  gStyle->SetPalette(1);
  pad->Divide(2, 2);
  pad->cd(1); tds->draw("yhat:rw");

  pad->cd(3); tds->draw("yhat");
  pad->cd(2); tds->draw("yhat:rw","","colz");

  pad->cd(4);
  // The parallel plot
  tds->draw("rw:r:tu:tl:hu:hl:l:kw:yhat","","para");
  TParallelCoord* para = (TParallelCoord*)gPad->GetListOfPrimitives()->FindObject(" ↩
      ParaCoord");

  // The output attribute
  TParallelCoordVar* axis = (TParallelCoordVar*)para->GetVarList()->FindObject("yhat");
  axis->AddRange(new TParallelCoordRange(axis,15.0,80.0));
  para->AddSelection("blue");
  para->GetCurrentSelection()->SetLineColor(kBlue);

}
```

The laws of distribution are set for each input variable. For example for the **Rw** variable, the macro creates an uniform distribution between bounds 0.05 and 0.15 associated to a `TAttribute` named **"rw"**:

```
tds->addAttribute( new TUniformDistribution("rw", 0.05,0.15));
```

Then, the `TAttribute` is linked to the input file with:

```
tds->getAttribute("rw")->setFileKey("flowrate_input_with_xml.in", "frottement_paroi/@rw","% ↩
    e", TAttributeFileKey::kXMLAttribute);
```

which will be understood by Uranie as the fact the **rw** variable has to be read in the input file with XML format as an XML attribute (`kXMLAttribute`) with the path **frottement_paroi/@rw**. In the same way, others `TAttributes` are added, some with XML fields:

```
tds->getAttribute("tu")->setFileKey(sIn, "parameter[@name='Tu']/value/double", "%e",  ↩
    TAttributeFileKey::kXMLField);
```

A design-of-experiments is then built with 100 samplings using the LHS method.

```
TSampling *sampling = new TSampling(tds, "lhs", 100);
sampling->generateSample();
```

We want the output file to be with XML format. So the output file `_output_flowrate_withXML_.dat` is set as a `TOutputFileXML`.

```
TOutputFileXML *fout = new TOutputFileXML("_output_flowrate_withXML_.dat");
```

The variable **yhat** and **d** could be linked to this `TOutputFileRow` output file. But here, only **yhat** will be considered as variable of interest by Uranie, by passing Uranie the new attribute, its XML path and type:

```
fout->addAttribute(new TAttribute("yhat"), "/steady_state[@name='flowrate']/parameter/value ↩
    /double", URANIE::DataServer::TAttributeFileKey::kXMLField);
```

We set the code as being the **flowrate** execution with *"-x"* option to deal with XML inputs.

```
TCode *mycode = new TCode(tds, "flowrate -x");
```

which indicates that **flowrate** has to find input files with XML format.

Then the launcher is initialised with a `TDataServer` and the code is executed:

```
TLauncher *lanceur = new TLauncher(tds, mycode);
```

The launcher will execute the **flowrate -k** command for each of the 100 patterns with the `run` method:

```
tlch->run();
```

XIV.4.6.3 Graph

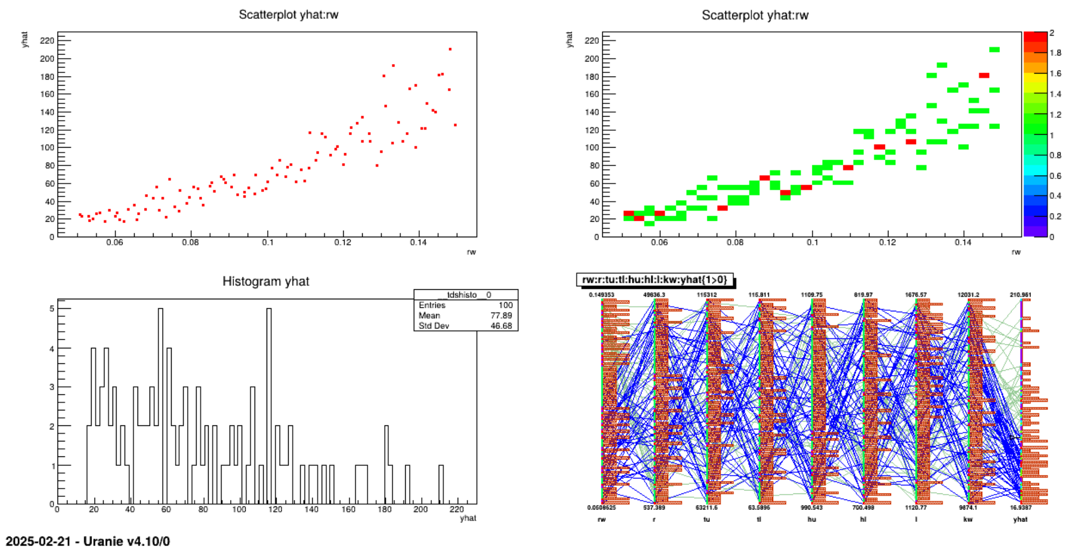


Figure XIV.22: Graph of the macro "`launchCodeFlowrateXMLSampling.C`"

XIV.4.7 Macro "`launchCodeFlowrateKeySamplingKey.C`"

XIV.4.7.1 Objective

The objective of the macro is to evaluate the **flowrate** external code on a design-of-experiments where input `file "flowrate_input_with_keys.in"` is with type "key=value" and the output file "`_output_flowrate_withKey_.dat`" is also of the "key=value" type. This code is described in Section IV.1.2.3.1.1.

The used input file `flowrate_input_with_keys.in` has "key=value" format:

```
#
#
# INPUT FILE with KEYS for the "FLOWREATE" code
# \date   2008-04-22 12:53:35
#


date = 123456 ;


#########################
##
```

```
##  exclude points
##
chu = 1050;
chl = 770;
cr  = 1100;
##
#########################

#########################
##
##  parameters : 8
##
Rw = 0.0500 ;
R = 33366.67 ;
Tu = 63070.0 ;
Tl = 116.00 ;
Hu = 1110.00 ;
Hl = 768.57;
L = 1200.0 ;
Kw = 11732.14 ;
##
#########################


#########################
##
## to simulate CPU time
##
## normal         1 :
## min     10000000 : 1.160u 0.000s 0:01.16 100.0%
## max    100000000 : 11.600u 0.010s 0:11.61 100.0%
##
nLoop = 1;
##
#########################

end = 6;
```

It defines the eight variables r_ω, r, T_u, T_l, H_u, H_l, L, K_ω and performs the execution of the command **flowrate -k**.

The output file `_output_flowrate_withKey_.dat` is with *"key=value"* type. It looks like:

```
yhat = 6.757218e+01;
d = 4.092561e+03;
```

yhat and **d** are defined as output variables in the macro.

XIV.4.7.2 Macro Uranie

```
void launchCodeFlowrateKeySamplingKey(Int_t nS = 100)
{

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsFlowrate", "Design of experiments for Flowrate");

  // Add the study attributes ( min, max and nominal values)
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
```

```
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

// The reference input file
TString sIn = TString("flowrate_input_with_keys.in");

// Set the reference input file and the key for each input attributes
tds->getAttribute("rw")->setFileKey(sIn, "Rw");
tds->getAttribute("r")->setFileKey(sIn, "R");
tds->getAttribute("tu")->setFileKey(sIn, "Tu");
tds->getAttribute("tl")->setFileKey(sIn, "Tl");
tds->getAttribute("hu")->setFileKey(sIn, "Hu");
tds->getAttribute("hl")->setFileKey(sIn, "Hl");
tds->getAttribute("l")->setFileKey(sIn, "L");
tds->getAttribute("kw")->setFileKey(sIn, "Kw");

// Generate the Design of Experiments
TSampling *sampling = new TSampling(tds, "lhs", nS);
sampling->generateSample();

// The output file of the code
TOutputFileKey *fout = new TOutputFileKey("_output_flowrate_withKey_.dat");
// The attribute in the output file
fout->addAttribute(new TAttribute("yhat"));
fout->addAttribute(new TAttribute("d"));

// Instanciation de mon code
TCode *mycode = new TCode(tds, "flowrate -s -k");
// on ajoute le fichier de sortie du code
mycode->addOutputFile( fout );

// Lancement du code
TLauncher *lanceur = new TLauncher(tds, mycode);
lanceur->setSave();
lanceur->setClean();
//lanceur->setWorkingDirectory(gSystem->Getenv("PWD") + TString("/tmpLanceurUranie/ ↩
    flowrate"));
lanceur->setVarDraw("yhat:rw","","");

lanceur->run();

//  tds->exportData("_flowrate_sampler_launcher_.dat");
tds->startViewer();

// Visualisation
TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro launchCodeFlowrateOATMinMax" ↩
    ,5,64,1270,667);
TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
gStyle->SetPalette(1);
pad->Divide(2, 2);
pad->cd(1); tds->draw("yhat:rw");

pad->cd(3); tds->draw("yhat");
pad->cd(2); tds->draw("yhat:rw","","colz");

pad->cd(4);
// The parallel plot
tds->draw("rw:r:tu:tl:hu:hl:l:kw:yhat","","para");
TParallelCoord* para = (TParallelCoord*)gPad->GetListOfPrimitives()->FindObject(" ↩
    ParaCoord");
```

```
  // The output attribute
  TParallelCoordVar* axis = (TParallelCoordVar*)para->GetVarList()->FindObject("yhat");
  axis->AddRange(new TParallelCoordRange(axis,15.0,20.0));
  para->AddSelection("blue");
  para->GetCurrentSelection()->SetLineColor(kBlue);

  axis->AddRange(new TParallelCoordRange(axis,155.0,160.0));
  para->AddSelection("blue");
  para->GetCurrentSelection()->SetLineColor(kBlue);

}
```

The laws of distributions are set for each input variable. For example for the **rw** variable, the macro creates an uniform distribution between bounds 0.05 and 0.15 associated to a `TAttribute` named **"rw"** (note that the set value in `flowrate_input_with_keys.in` for **Rw**, 0.05, is between bounds. It is important to check this property when creating the distribution):

```
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
```

Then, the `TAttribute` is linked to the input file with:

```
tds->getAttribute("rw")->setFileKey(TString("flowrate_input_with_keys.in"), "Rw");
```

which will be understood by Uranie as the fact the **rw** variable has to be read in an input file with "key=value" format with the key **Rw**.

A design-of-experiments is then built with 100 samplings using the LHS method.

```
TSampling *sampling = new TSampling(tds, "lhs", 100);
sampling->generateSample();
```

The output file with "key=value" type of format `_output_flowrate_withKey_.dat` is instantiated as a `TOutputFil`

```
TOutputFileKey *fout = new TOutputFileKey("_output_flowrate_withKey_.dat");
```

The two output variables **yhat** and **d** are then linked to this `TOutputFileKey` output file.

```
fout->addAttribute(new TAttribute("d"));
fout->addAttribute(new TAttribute("yhat"));
```

We set the code as being the **flowrate** execution with *"-k"*option:

```
TCode *mycode = new TCode(tds, "flowrate -k");
```

in which the "-k" option indicates that **flowrate** code has to find "key=value" input files.

The launcher is initialised:

```
TLauncher *tlch = new TLauncher(tds, mycode);
```

The launcher will execute the **flowrate -k** command for each of the 100 patterns and launched with the `run` method:

```
tlch->run();
```

XIV.4.7.3 Graph

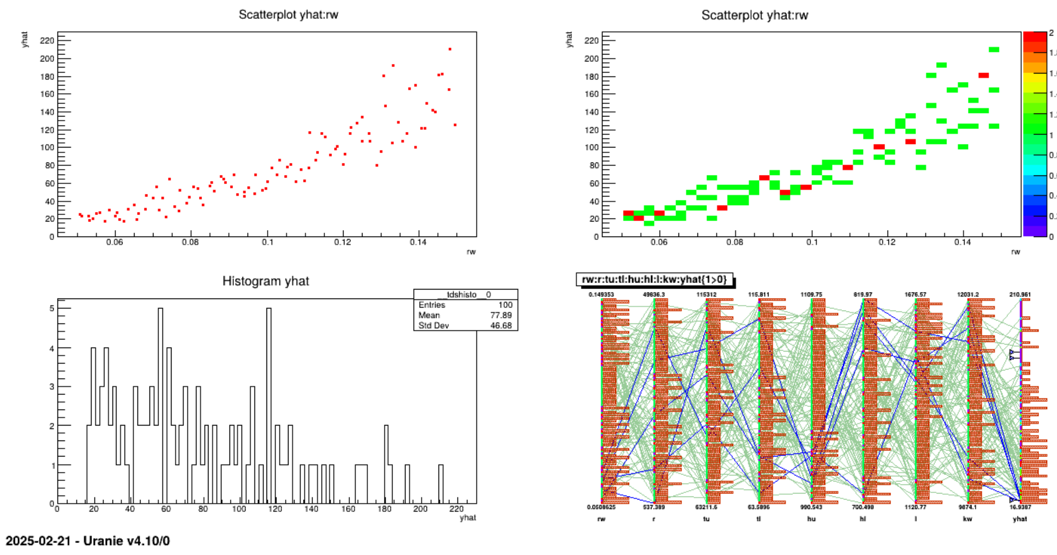


Figure XIV.23: Graph of the macro "`launchCodeFlowrateKeySamplingKey.C`"

XIV.4.8 Macro "`launchCodeFlowrateKeyRecreateSampling.C`"

XIV.4.8.1 Objective

The objective of the macro is to evaluate the **flowrate** external code on a design-of-experiments where the input file "`flowrate_input_with_keys.in`" which is produced by Uranie on the fly is with type "key=value" and the output file "`_output_flowrate_withKey_.dat`" with type "key=value". This **flowrate** code is described in Section IV.1.2.3.1.1.

The used input file `flowrate_input_with_keys.in` has "key=value" format and will be produced on the fly by the code:

```
Rw = 1.480238e-01 ;
R = 3.644187e+03 ;
Tu = 7.840305e+04 ;
Tl = 8.467102e+01 ;
Hu = 1.008708e+03 ;
Hl = 7.667991e+02 ;
L = 1.174857e+03 ;
Kw = 1.097634e+04 ;
```

This file defines the eight variables r_ω, r, T_u, T_l, H_u, H_l, L, K_ω needed to perform the execution of the command **flowrate -k**.

The output file `_output_flowrate_withRow_.dat` is with "values in rows" type. It looks like:

```
#COLUMN_NAMES: yhat | d

6.757218e+01  4.092561e+03
```

yhat and **d** are defined as output variables in the macro. But here, only **yhat** is considered as variable of interest by Uranie.

XIV.4.8.2 Macro Uranie

```
void launchCodeFlowrateKeyRecreateSampling(Int_t nS = 100)
{

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsFlowrate", "Design of experiments for Flowrate");

  // Add the study attributes ( min, max and nominal values)
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  // The reference input file
  TString sIn = TString("flowrate_input_with_keys.in");

  // Set the reference input file and the key for each input attributes
  tds->getAttribute("rw")->setFileKey(sIn, "Rw","",TAttributeFileKey::kNewKey);
  tds->getAttribute("r")->setFileKey(sIn, "R","",TAttributeFileKey::kNewKey);
  tds->getAttribute("tu")->setFileKey(sIn, "Tu","",TAttributeFileKey::kNewKey);
  tds->getAttribute("tl")->setFileKey(sIn, "Tl","",TAttributeFileKey::kNewKey);
  tds->getAttribute("hu")->setFileKey(sIn, "Hu","",TAttributeFileKey::kNewKey);
  tds->getAttribute("hl")->setFileKey(sIn, "Hl","",TAttributeFileKey::kNewKey);
  tds->getAttribute("l")->setFileKey(sIn, "L","",TAttributeFileKey::kNewKey);
  tds->getAttribute("kw")->setFileKey(sIn, "Kw","",TAttributeFileKey::kNewKey);

  // Generate the Design of Experiments
  TSampling *sampling = new TSampling(tds, "lhs", nS);
  sampling->generateSample();

  // The output file of the code
  TOutputFileKey *fout = new TOutputFileKey("_output_flowrate_withKey_.dat");
  // The attribute in the output file
  fout->addAttribute(new TAttribute("yhat"));

  // Instanciation de mon code
  TCode *mycode = new TCode(tds, "flowrate -s -k");
  // on ajoute le fichier de sortie du code
  mycode->addOutputFile( fout );

  // Lancement du code
  TLauncher *lanceur = new TLauncher(tds, mycode);
  lanceur->setSave();
  lanceur->setClean();
  //lanceur->setWorkingDirectory(gSystem->Getenv("PWD") + TString("/tmpLanceurUranie/ ↩
      flowrate"));
  lanceur->setVarDraw("yhat:rw","","");

  lanceur->run();



  tds->exportData("_flowrate_sampler_launcher_.dat");

   // Visualisation
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro  ↩
      launchCodeFlowrateKeyRecreateSampling",5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  gStyle->SetPalette(1);
  pad->Divide(2, 2);
  pad->cd(1); tds->draw("yhat:rw");

  pad->cd(3); tds->draw("yhat");
  pad->cd(2); tds->draw("yhat:rw","","colz");

  pad->cd(4);
  // The parallel plot
  tds->draw("rw:r:tu:tl:hu:hl:l:kw:yhat","","para");
  TParallelCoord* para = (TParallelCoord*)gPad->GetListOfPrimitives()->FindObject(" ↩
      ParaCoord");

  // The output attribute
  TParallelCoordVar* axis = (TParallelCoordVar*)para->GetVarList()->FindObject("yhat");
  axis->AddRange(new TParallelCoordRange(axis,15.0,80.0));
  para->AddSelection("blue");
  para->GetCurrentSelection()->SetLineColor(kBlue);

}
```

In the first part of this macro, properties are set for each input variable. For example for the **Rw** variable, the macro creates an uniform distribution between bounds 0.05 and 0.15 associated to a `TAttribute` named **"rw"**:

```
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
```

Then, the `TAttribute` is linked to the input file with:

```
tds->getAttribute("rw")->setFileKey(TString("flowrate_input_keys.in"), "Rw","", ↩
    TAttributeFileKey::kNewKey);
```

which will be understood by Uranie as the fact the **rw** variable has to be read in the input file with "key=value" format with the key **Rw**. The property `TAttributeFileKey::kNewKey` means that the input file is of "key=value" type and has to be created on the fly for each sampling locations.

A design-of-experiments is then built with 100 samplings using the LHS method.

```
TSampling *sampling = new TSampling(tds, "lhs", 100);
sampling->generateSample();
```

The output file with "values in rows" type `_output_flowrate_withRow_.dat` is instantiated set as a `TOutputFileRow`.

```
TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
```

The variable **yhat** and **d** could be linked to this `TOutputFileRow` output file. But here, only **yhat** will be considered by Uranie as variable of interest.

```
fout->addAttribute("yhat");
```

We set the code as being the **flowrate** execution with *"-k"* option.

```
TCode *mycode = new TCode(tds, "flowrate -k");
```

which indicates that **flowrate** code has to find "key=value" input files.

Then the launcher is initialised with a `TDataServer` and the code is executed:

```
TLauncher *tlch = new TLauncher(tds, mycode);
```

The launcher will execute the **flowrate -k** command for each of the 100 patterns with the `run` method:

```
tlch->run();
```

XIV.4.8.3 Graph

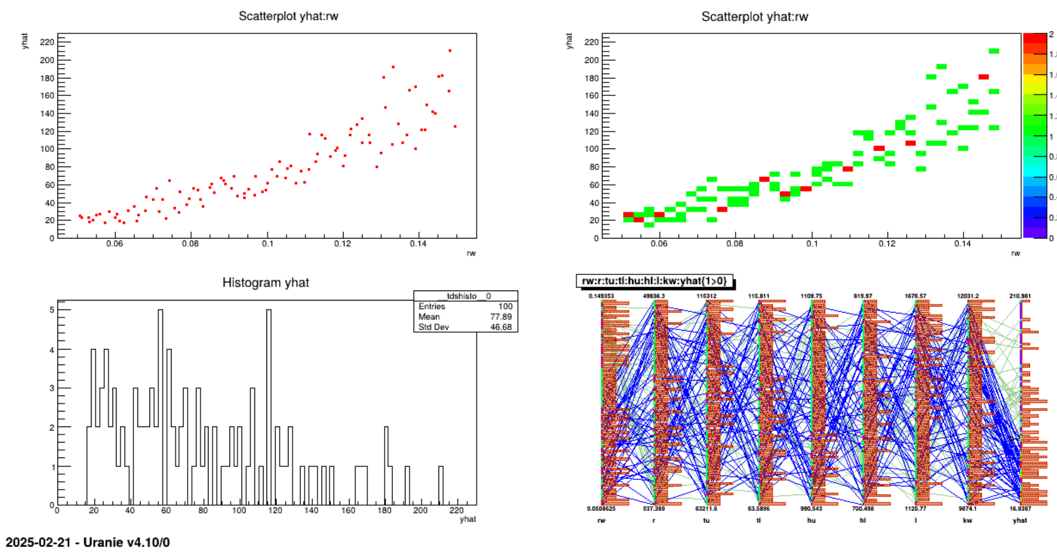


Figure XIV.24: Graph of the macro "`launchCodeFlowrateKeyRecreateSampling.C`"

XIV.4.9 Macro "`launchCodeFlowrateKeyRecreateSamplingOutputDataServer.C`"

XIV.4.9.1 Objective

The objective of the macro is to evaluate the **flowrate** external code on a design-of-experiments where the input file "`flowrate_input_with_keys.in`" which will be produced by Uranie on the fly is with type "key=value" and the output file "`_output_flowrate_withRow_.dat`" is with type "DataServer". This **flowrate** code is described in Section IV.1.2.3.1.1.

The input file `flowrate_input_with_keys.in` has "key=value" format and will be produced on the fly by the code:

```
#
#
# INPUT FILE with KEYS for the "FLOWREATE" code
# \date   2008-04-22 12:53:35
#


date = 123456 ;


#########################
##
##  exclude points
##
chu = 1050;
chl = 770;
cr  = 1100;
##
#########################

#########################
```

```
##
##  parameters : 8
##
Rw = 0.0500 ;
R = 33366.67 ;
Tu = 63070.0 ;
Tl = 116.00 ;
Hu = 1110.00 ;
Hl = 768.57;
L = 1200.0 ;
Kw = 11732.14 ;
##
#########################


#########################
##
## to simulate CPU time
##
## normal          1 :
## min      10000000 : 1.160u 0.000s 0:01.16 100.0%
## max     100000000 : 11.600u 0.010s 0:11.61 100.0%
##
nLoop = 1;
##
#########################

end = 6;
```

This file defines the eight variables r_ω, r, T_u, T_l, H_u, H_l, L, K_ω needed to perform the execution of the command **flowrate -k**.

The output file _output_flowrate_withRow_.dat is with "DataServer" type. It looks like:

```
#COLUMN_NAMES: yhat | d

6.757218e+01  4.092561e+03
```

yhat and **d** are defined as output variables in the macro. But here, only **yhat** is considered as variable of interest by Uranie.

XIV.4.9.2 Macro Uranie

```
void launchCodeFlowrateKeyRecreateSamplingOutputDataServer(Int_t nS = 100)
{

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsFlowrate", "Design of experiments for Flowrate");

  // Add the study attributes ( min, max and nominal values)
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

```
// The reference input file
TString sIn = TString("flowrate_input_with_keys.in");

// Set the reference input file and the key for each input attributes
tds->getAttribute("rw")->setFileKey(sIn, "Rw","",TAttributeFileKey::kNewKey);
tds->getAttribute("r")->setFileKey(sIn, "R","",TAttributeFileKey::kNewKey);
tds->getAttribute("tu")->setFileKey(sIn, "Tu","",TAttributeFileKey::kNewKey);
tds->getAttribute("tl")->setFileKey(sIn, "Tl","",TAttributeFileKey::kNewKey);
tds->getAttribute("hu")->setFileKey(sIn, "Hu","",TAttributeFileKey::kNewKey);
tds->getAttribute("hl")->setFileKey(sIn, "Hl","",TAttributeFileKey::kNewKey);
tds->getAttribute("l")->setFileKey(sIn, "L","",TAttributeFileKey::kNewKey);
tds->getAttribute("kw")->setFileKey(sIn, "Kw","",TAttributeFileKey::kNewKey);

// Generate the design of experiments
TSampling *sampling = new TSampling(tds, "lhs", nS);
sampling->generateSample();

// The output file of the code
TOutputFileDataServer *fout = new TOutputFileDataServer("_output_flowrate_withRow_.dat");
// The attribute in the output file
fout->addAttribute("yhat"); // fout->addAttribute(new TAttribute("y"));

// Create a TCode object from a TDS and the command line
TCode *mycode = new TCode(tds, "flowrate -s -k");
// Add the output file
mycode->addOutputFile( fout );

// Create a launcher of code
TLauncher *lanceur = new TLauncher(tds, mycode);
lanceur->setSave();
lanceur->setClean();
//lanceur->setWorkingDirectory(gSystem->Getenv("PWD") + TString("/tmpLanceurUranie/ ↩
    flowrate"));

lanceur->setVarDraw("yhat:rw","","");

lanceur->run();



tds->exportData("_flowrate_sampler_launcher_.dat");
// Visualisation
TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro  ↩
    launchCodeFlowrateKeyRecreateSamplingOutputDataServer",565,62,560,619);
TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
gStyle->SetPalette(1);
pad->Divide(1, 3);

// Scatterplot y versus rw
pad->cd(1); tds->draw("yhat:rw");

// Histogramm of the output attribute
pad->cd(2); tds->draw("yhat");

// The parallel plot
pad->cd(3);
tds->draw("rw:r:tu:tl:hu:hl:l:kw:yhat","","para");
TParallelCoord* para = (TParallelCoord*)gPad->GetListOfPrimitives()->FindObject(" ↩
    ParaCoord");

// The output attribute
TParallelCoordVar* axis = (TParallelCoordVar*)para->GetVarList()->FindObject("yhat");
```

```
    axis->AddRange(new TParallelCoordRange(axis,15.0,20.0));
    para->AddSelection("blue");
    para->GetCurrentSelection()->SetLineColor(kBlue);

    axis->AddRange(new TParallelCoordRange(axis,155.0,160.0));
    para->AddSelection("blue");
    para->GetCurrentSelection()->SetLineColor(kBlue);

}
```

The laws of distributions are set for each input variable. For example for the **rw** variable, the macro creates an uniform distribution between bounds 0.05 and 0.15 associated to a `TAttribute` named **"rw"**:

```
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
```

Then, the `TAttribute` is linked to the input file with:

```
tds->getAttribute("rw")->setFileKey(TString("flowrate_input_with_rows.in"), "Rw","", ↩
    TAttributeFileKey::kNewKey);
```

which will be understood by Uranie as the fact the **rw** variable has to be read in an input file with "key=value" format. The property `TAttributeFileKey::kNewKey` means the input file has to be created on the fly for each sampling created.

A design-of-experiments is then built with 100 samplings using the LHS method.

```
TSampling *sampling = new TSampling(tds, "lhs", 100);
sampling->generateSample();
```

The output file with "DataServer" type `_output_flowrate_withRow_.dat` is instantiated as a `TOutputFileDataServe`
This `TOutputFileDataServer` file is of the same format that a `TDataServer` can load.

```
TOutputFileDataServer *fout = new TOutputFileDataServer("_output_flowrate_withRow_.dat");
```

The variable **yhat** and **d** are then linked to this `TOutputFileDataServer` output file. But here, only **yhat** will be considered by Uranie as a variable of interest.

```
fout->addAttribute("yhat");
```

We set the code as being the **flowrate** execution with *"-k"* option.

```
TCode *mycode = new TCode(tds, "flowrate -k");
```

which indicates that **flowrate** code has to find "key=value" input files.

Then the launcher is initialised with a `TDataServer` and the code is executed:

```
TLauncher *lanceur = new TLauncher(tds, mycode);
```

The launcher will execute the **flowrate -k** command for each of the 100 patterns with the `run` method.

```
lanceur->run();
```

XIV.4.9.3 Graph

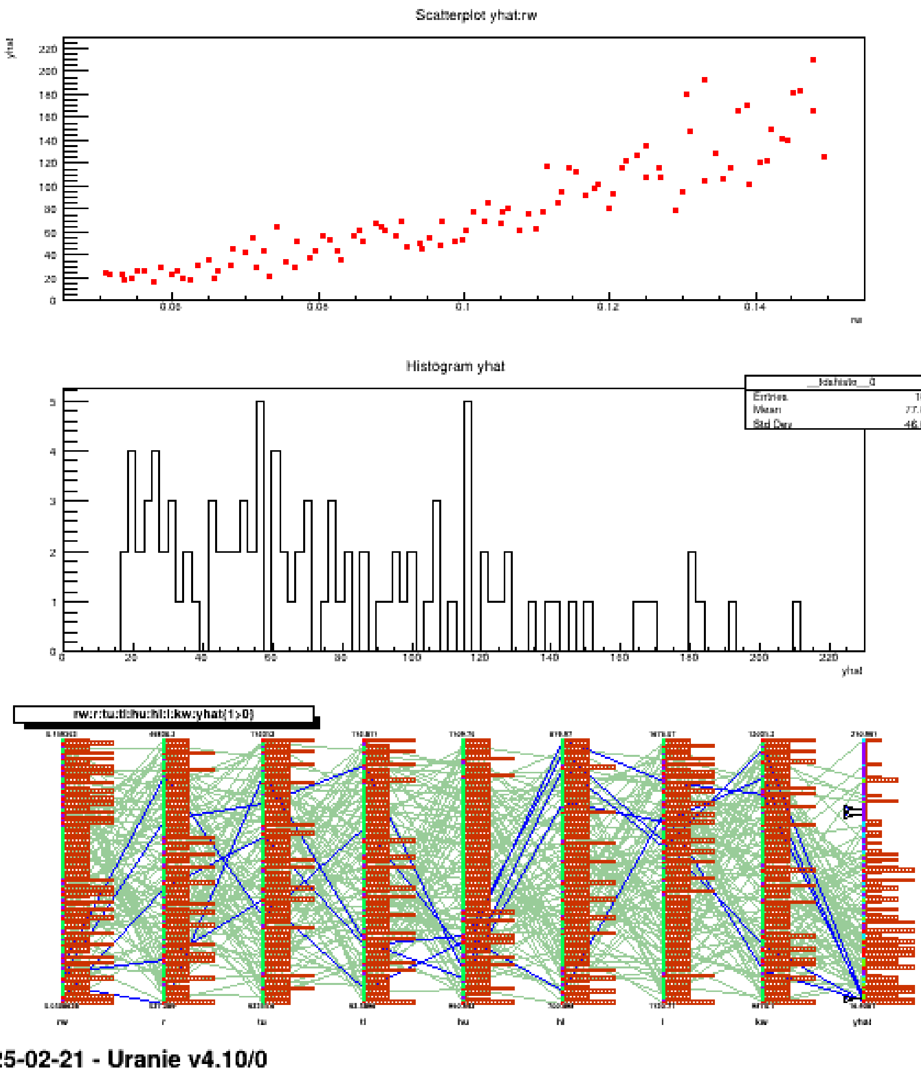


Figure XIV.25: Graph of the macro "`launchCodeFlowrateKeyRecreateSamplingOutputDataServer.C`"

XIV.4.10 Macro "`launchCodeFlowrateRowRecreateSamplingOutputDataServer.C`"

XIV.4.10.1 Objective

The objective of the macro is to evaluate the **flowrate** external code on a design-of-experiments where the input file "`flowrate_input_with_values_rows.in`" will be produced on the fly by Uranie is with "values in rows" type and the output file "`_output_flowrate_withRow_.dat`" with "DataServer" type. This **flowrate** code is described in Section IV.1.2.3.1.1.

The input file `flowrate_input_with_values_rows.in` is with "values in rows" format and will be produced on the fly by the code:

```
0.0500 33366.67 63070.0 116.00 1110.00 768.57 1200.0 11732.14
```

This file defines the eight variables r_ω, r, T_u, T_l, H_u, H_l, L, K_ω needed to perform the execution of the command **flowrate -r**.

The output file `_output_flowrate_withRow_.dat` is of "DataServer" type. It looks like:

```
#COLUMN_NAMES: yhat | d

6.757218e+01  4.092561e+03
```

yhat and **d** are defined as output variables in the macro.

XIV.4.10.2 Macro Uranie

```
void launchCodeFlowrateRowRecreateSamplingOutputDataServer(Int_t nS = 100)
{

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsFlowrate", "Design of experiments for Flowrate");

  // Add the study attributes ( min, max and nominal values)
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  // The input file
  TString sFileName = TString("flowrate_input_with_values_rows.in");

  // Set the input file with "key = value" format CREATED ON FLY with row for each input ↩
      attributes
  tds->getAttribute("rw")->setFileKey(sFileName, "Rw","",TAttributeFileKey::kNewRow);
  tds->getAttribute("r")->setFileKey(sFileName, "R","",TAttributeFileKey::kNewRow);
  tds->getAttribute("tu")->setFileKey(sFileName, "Tu","",TAttributeFileKey::kNewRow);
  tds->getAttribute("tl")->setFileKey(sFileName, "Tl","",TAttributeFileKey::kNewRow);
  tds->getAttribute("hu")->setFileKey(sFileName, "Hu","",TAttributeFileKey::kNewRow);
  tds->getAttribute("hl")->setFileKey(sFileName, "Hl","",TAttributeFileKey::kNewRow);
  tds->getAttribute("l")->setFileKey(sFileName, "L","",TAttributeFileKey::kNewRow);
  tds->getAttribute("kw")->setFileKey(sFileName, "Kw","",TAttributeFileKey::kNewRow);

  // Generate a Design of experiments (DoE) from the TDataServer
  TSampling *sampling = new TSampling(tds, "lhs", nS);
  sampling->generateSample();

  // The output file of the code based on the same format that a TDataServer can load (# ↩
      COLUMN_NAMES: yhat | d)
  TOutputFileDataServer *fout = new TOutputFileDataServer("_output_flowrate_withRow_.dat");
  // The attributes in the output file
  fout->addAttribute("d"); // fout->addAttribute(new TAttribute("d"));
  fout->addAttribute("yhat"); // fout->addAttribute(new TAttribute("yhat"));

  // Create a TCode object with the TDS (attribute and input files) and the command to ↩
      execute
  TCode *mycode = new TCode(tds, "flowrate -s -r");
  // Add the output file
  mycode->addOutputFile( fout );
  // Launcher of the code on the Design of experiments (DoE) in the TDS
  TLauncher *tlch = new TLauncher(tds, mycode);
  tlch->setSave();
  tlch->setClean();
  //tlch->setWorkingDirectory(gSystem->Getenv("PWD") + TString("/tmpLanceurUranie/flowrate ↩
      "));
  tlch->setVarDraw("yhat:rw","","");

  // Evaluate the DoE
  tlch->run();



//   tds->exportData("_flowrate_sampler_launcher_.dat");

  // Visualisation
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro  ↩
      launchCodeFlowrateRowRecreateSamplingOutputDataServer.C",565,62,560,619);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  gStyle->SetPalette(1);
  pad->Divide(1, 3);
  pad->cd(1); tds->draw("yhat:rw");

  pad->cd(2);
  // Histogramm of the output attribute
  tds->draw("yhat");

  pad->cd(3);
  // The parallel plot
  tds->draw("rw:r:tu:tl:hu:hl:l:kw:yhat","","para");
  TParallelCoord* para = (TParallelCoord*)gPad->GetListOfPrimitives()->FindObject(" ↩
      ParaCoord");

  // The output attribute
  TParallelCoordVar* axis = (TParallelCoordVar*)para->GetVarList()->FindObject("yhat");
  axis->AddRange(new TParallelCoordRange(axis,15.0,20.0));
  para->AddSelection("blue");

  para->GetCurrentSelection()->SetLineColor(kRed);
  axis->AddRange(new TParallelCoordRange(axis,155.0,160.0));

}
```

The laws of distribution are set for each input variable. For example for the **rw** variable, the macro creates an uniform distribution between bounds 0.05 and 0.15 associated to a `TAttribute` named **"rw"**:

```
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
```

Then, the `TAttribute` is linked to the input file with:

```
tds->getAttribute("rw")->setFileKey(TString("flowrate_input_with_rows.in"), "Rw","", ↩
    TAttributeFileKey::kNewRow);
```

which will be understood by Uranie as the fact the **rw** variable has to be read in an input file with "values in rows" format. The property `TAttributeFileKey::kNewRow` means the input file has to be created for each sampling created.

A design-of-experiments is then built with 100 samplings using the LHS method.

```
TSampling *sampling = new TSampling(tds, "lhs", 100);
sampling->generateSample();
```

The output file with "DataServer" type `_output_flowrate_withRow_.dat` is instantiated as a `TOutputFileDataServe`
This `TOutputFileDataServer` file is of the same format that a `TDataServer` can load.

```
TOutputFileDataServer *fout = new TOutputFileDataServer("_output_flowrate_withRow_.dat");
```

The variable **yhat** and **d** are then linked to this `TOutputFileDataServer` output file.

```
fout->addAttribute("d");
fout->addAttribute("yhat");
```

We set the code as being the **flowrate** execution with *"-r"* option:

```
TCode *mycode = new TCode(tds, "flowrate -r");
```

in which the "-r" option indicates that **flowrate** code has to find "values in rows" input files.

Then the launcher is initialised with a `TDataServer` and the code is executed:

```
TLauncher *tlch = new TLauncher(tds, mycode);
```

The launcher will execute the **flowrate -r** command for each of the 100 patterns with the `run` method.

```
tlch->run();
```

XIV.4.10.3 Graph

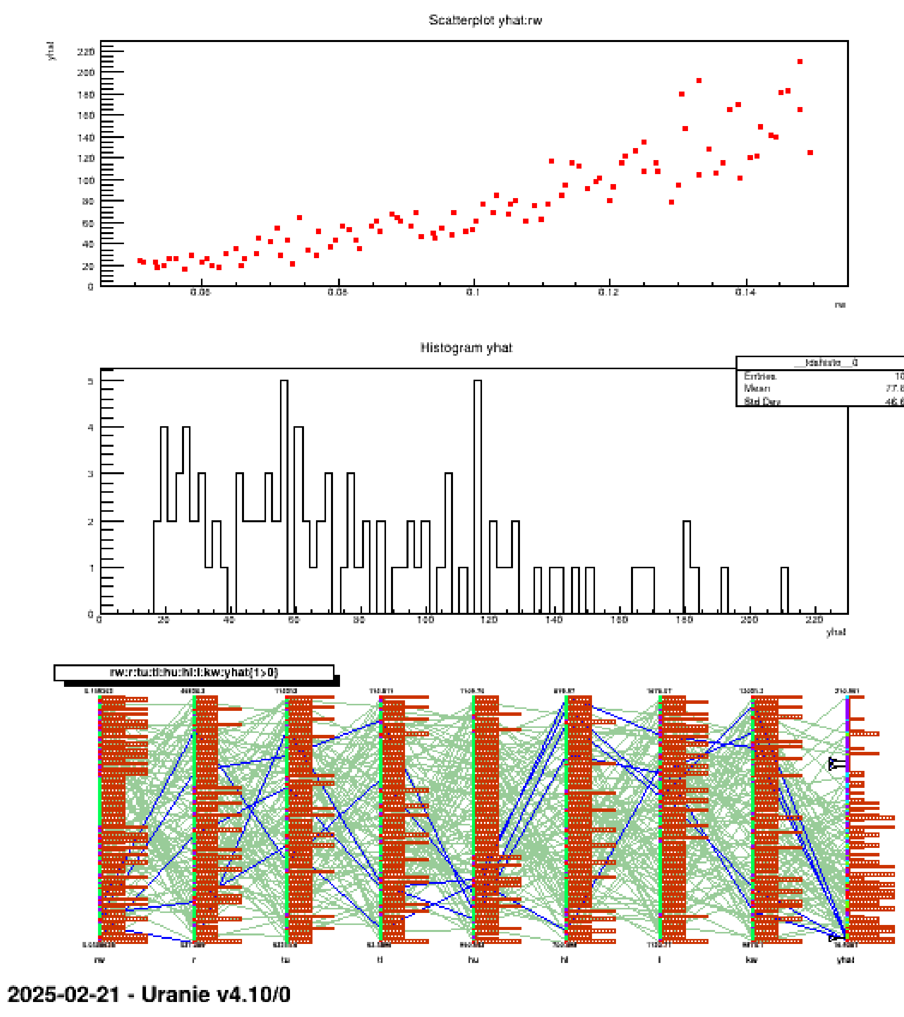


Figure XIV.26: Graph of the macro "launchCodeFlowrateRowRecreateSamplingOutputDataServer.C"

XIV.4.11 Macro "launchCodeFlowrateFlagSampling.C"

XIV.4.11.1 Objective

The objective of the macro is to evaluate the **flowrate** external code on the design-of-experiments where the input file "flowrate_input_with_flags.in" has flags" replaced with usable values and the output file "_output_flowrate_withRow_.dat" with "values in rows" type. This code **flowrate** is described in Section IV.1.2.3.1.1.

The creation of an input file containing some "flags" is motivated by the will to make evolve variables in an input file with no particular structure:

```
#
# INPUT FILE with FLAG for the "FLOWREATE" code
# \date   2008-04-22 12:55:17
#

new Implicit_Steady_State sch {
    frottement_paroi { 0.071623 19712.541454 }       // values of Rw and R
    tinit               0.0
    tmax                1000000.
    nb_pas_dt_max       1500
    dt_min              1051.972855                  // value of Hu
    dt_max              805.249178                   // value of Hl
    facsec              1000000.
    kW                  11401.611060                 // value of Kw
    information_Tu      Champ_Uniforme 1        85927.853162    // value of Tu
    information_Tl      Champ_Uniforme 1        85.803614       // value of Tl
    information_L {
        precision  1162.689830                       // value of L
    }
    convergence {
        criterion relative_max_du_dt
        precision  1.e-6
    }

    stop_criterium {
        ch_abcsissa_hu 1050
        ch_ordinate_hl 770
        c_radius 1100
    }

    Solveur Newton3 {
        max_iter_matrice        1
        max_iter_implicite      1
        date                    5654321
        seuil_convg_implicite   1.e-6
        assemblage_implicite    10
        solveur_lineaire        BiCGS
            preconditionneur    ILU
            seuil_resol_implicite 1.e-5
    }
}
```

These values are then replaced by "flags" bounded with special characters.

The input file `flowrate_input_with_flags.in` that contains "flags" is built this way:

```
#
# INPUT FILE with FLAG for the "FLOWREATE" code
# \date   2008-04-22 12:55:17
#

new Implicit_Steady_State sch {
    frottement_paroi { @Rw@ @R@ }
    tinit               0.0
    tmax                1000000.
    nb_pas_dt_max       1500
    dt_min              @Hu@
    dt_max              @Hl@
    facsec              1000000.
    kW                  @Kw@
    information_Tu      Champ_Uniforme 1        @Tu@
    information_Tl      Champ_Uniforme 1        @Tl@
    information_L {
        precision  @L@
    }
    convergence {
        criterion relative_max_du_dt
        precision  @Rw@
    }
    stop_criterium {
        ch_abcsissa_hu 1050
        ch_ordinate_hl 770
        c_radius 1100
    }

    Solveur Newton3 {
        max_iter_matrice        1
        max_iter_implicite      1
        date                    5654321
        seuil_convg_implicite   1.e-6
        assemblage_implicite    10
        solveur_lineaire        BiCGS
            preconditionneur    ILU
            seuil_resol_implicite 1.e-5
    }
}
```

This file defines the eight variables r_ω, r, T_u, T_l, H_u, H_l, L, K_ω needed to perform the execution of the command **flowrate -f**.

The output file `_output_flowrate_withRow_.dat` is of "values in rows" type. It looks like:

```
#COLUMN_NAMES: yhat | d

6.757218e+01  4.092561e+03
```

yhat and **d** are defined as output variables in the macro. But here, only **yhat** is considered as variable of interest by Uranie.

XIV.4.11.2 Macro Uranie

```
void launchCodeFlowrateFlagSampling(Int_t nS = 400)
{

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsFlowrate", "Design of experiments for Flowrate");

  // Add the study attributes ( min, max and nominal values)
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  // The reference input file
  TString sIn = TString("flowrate_input_with_flags.in");

  // Set the reference input file and the key for each input attributes
  tds->getAttribute("rw")->setFileFlag(sIn, "@Rw@");
  tds->getAttribute("r")->setFileFlag(sIn, "@R@");
  tds->getAttribute("tu")->setFileFlag(sIn, "@Tu@");
  tds->getAttribute("tl")->setFileFlag(sIn, "@Tl@");
  tds->getAttribute("hu")->setFileFlag(sIn, "@Hu@");
  tds->getAttribute("hl")->setFileFlag(sIn, "@Hl@");
  tds->getAttribute("l")->setFileFlag(sIn, "@L@");
  tds->getAttribute("kw")->setFileFlag(sIn, "@Kw@");

  // Generate the Design of Experiments
  TSampling *sampling = new TSampling(tds, "lhs", nS);
  sampling->generateSample();

  // The output file of the code
  TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
  // The attribute in the output file
  fout->addAttribute(new TAttribute("yhat"));


  // Instanciation de mon code
  TCode *mycode = new TCode(tds, "flowrate -s -f");
  // on ajoute le fichier de sortie du code
  mycode->addOutputFile( fout );

  // Lancement du code
  TLauncher *lanceur = new TLauncher(tds, mycode);
  lanceur->setSave();
  //  lanceur->setClean();
  //lanceur->setWorkingDirectory(gSystem->Getenv("PWD") + TString("/tmpLanceurUranie/ ↩
      flowrate"));
  lanceur->setVarDraw("yhat:rw","","");

  lanceur->run();



  tds->exportData("_flowrate_sampler_launcher_.dat");

  // Visualisation
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro launchCodeFlowrateFlag" ↩
      ,5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  gStyle->SetPalette(1);
  pad->Divide(2, 2);
  pad->cd(1); tds->draw("yhat:rw");

  pad->cd(3); tds->draw("yhat");
  pad->cd(2); tds->draw("yhat:rw","","colz");

  pad->cd(4);
  // The parallel plot
  tds->draw("rw:r:tu:tl:hu:hl:l:kw:yhat","","para");
  TParallelCoord* para = (TParallelCoord*)gPad->GetListOfPrimitives()->FindObject(" ↩
      ParaCoord");

  // The output attribute
  TParallelCoordVar* axis = (TParallelCoordVar*)para->GetVarList()->FindObject("yhat");
  axis->AddRange(new TParallelCoordRange(axis,100.0,160.0));
  para->AddSelection("blue");
  para->GetCurrentSelection()->SetLineColor(kBlue);

}
```

The laws of distribution are set for each input variable. For example for the **rw** variable, the macro creates an uniform distribution between bounds 0.05 and 0.15 associated to a `TAttribute` named **"rw"**:

```
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
```

Then, the `TAttribute` is linked to the input file with:

```
tds->getAttribute("rw")->setFileFlag(TString("flowrate_input_with_flags.in"), "@Rw@");
```

which will be understood by Uranie as the fact the **rw** variable has to be read in an input file with "flags".

A design-of-experiments is then built with 100 samplings using the LHS method.

```
TSampling *sampling = new TSampling(tds, "lhs", 100);
sampling->generateSample();
```

The output file with "values in rows" type `_output_flowrate_withRow_.dat` is instantiated as a `TOutputFileRow`

```
TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
```

The variable **yhat** and **d** are then linked to this `TOutputFileRow` output file. But here, only **yhat** will be considered as variable of interest by Uranie.

```
fout->addAttribute("yhat");
```

We set the code as being the **flowrate** execution with *"-f"* option.

```
TCode *mycode = new TCode(tds, "flowrate -f");
```

which indicates that **flowrate** code has to find input files with "flags". Then the launcher is initialised with a `TDataServer` and the code is executed:

```
TLauncher *tlch = new TLauncher(tds, mycode);
```

The launcher will execute the **flowrate -f** command for each of the 100 patterns with the `run` method:

```
tlch->run();
```

XIV.4.11.3 Graph

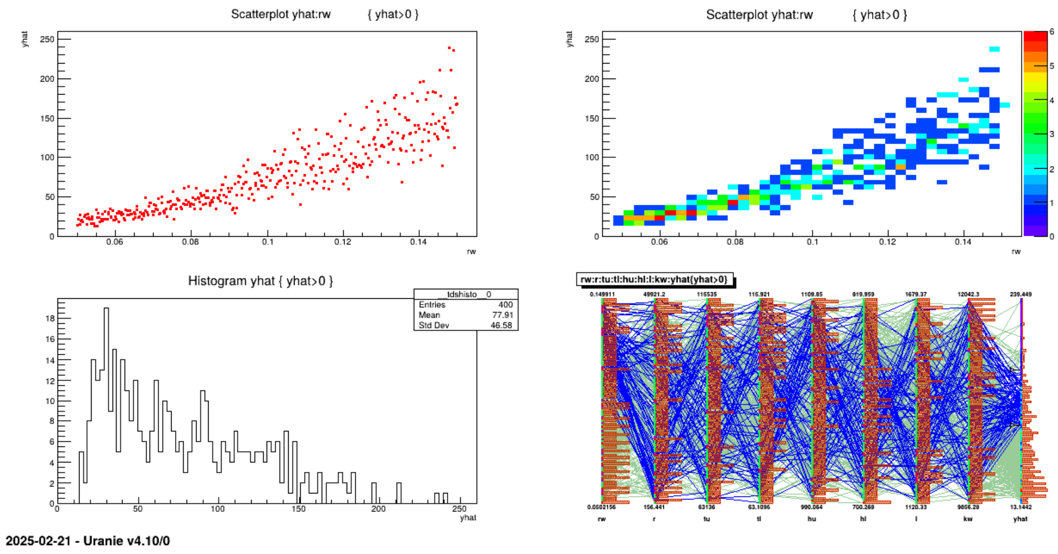


Figure XIV.27: Graph of the macro "`launchCodeFlowrateFlagSampling.C`"

XIV.4.12 Macro "launchCodeFlowrateFlagSamplingKey.C"

XIV.4.12.1 Objective

The objective of the macro is to evaluate the **flowrate** external code on a design-of-experiments where the input file "flowrate_input_with_flags.in" has "flags" and the output file "_output_flowrate_withKey_.dat" is with type "key=value". This code is described in Section IV.1.2.3.1.1.

The creation of an input file containing some "flags" is motivated by the will to make evolve variables in an input file with no particular structure:

```
#
# INPUT FILE with FLAG for the "FLOWREATE" code
# \date   2008-04-22 12:55:17
#

new Implicit_Steady_State sch {
    frottement_paroi { 0.071623 19712.541454 }       // values of Rw and R
    tinit               0.0
    tmax                1000000.
    nb_pas_dt_max       1500
    dt_min              1051.972855                  // value of Hu
    dt_max              805.249178                   // value of Hl
    facsec              1000000.
    kW                  11401.611060                 // value of Kw
    information_Tu      Champ_Uniforme 1        85927.853162    // value of Tu
    information_Tl      Champ_Uniforme 1        85.803614       // value of Tl
    information_L {
        precision  1162.689830                       // value of L
    }
    convergence {
        criterion relative_max_du_dt
        precision  1.e-6
    }

    stop_criterium {
        ch_abcsissa_hu 1050
        ch_ordinate_hl 770
        c_radius 1100
    }

    Solveur Newton3 {
        max_iter_matrice        1
        max_iter_implicite      1
        date                    5654321
        seuil_convg_implicite   1.e-6
        assemblage_implicite    10
        solveur_lineaire        BiCGS
            preconditionneur    ILU
            seuil_resol_implicite 1.e-5
    }
}
```

These values are then replaced by "flags" bounded with special characters.

The input file flowrate_input_with_flags.in containing "flags" is built this way:

```
#
# INPUT FILE with FLAG for the "FLOWREATE" code
# \date   2008-04-22 12:55:17
#
```

```
new Implicit_Steady_State sch {
    frottement_paroi { @Rw@ @R@ }
    tinit                0.0
    tmax                 1000000.
    nb_pas_dt_max        1500
    dt_min               @Hu@
    dt_max               @Hl@
    facsec               1000000.
    kW                   @Kw@
    information_Tu       Champ_Uniforme 1         @Tu@
    information_Tl       Champ_Uniforme 1         @Tl@
    information_L {
        precision  @L@
    }
    convergence {
        criterion relative_max_du_dt
        precision  @Rw@
    }

    stop_criterium {
        ch_abcsissa_hu 1050
        ch_ordinate_hl 770
        c_radius 1100
    }

    Solveur Newton3 {
        max_iter_matrice        1
        max_iter_implicite      1
        date                    5654321
        seuil_convg_implicite   1.e-6
        assemblage_implicite    10
        solveur_lineaire        BiCGS
            preconditionneur    ILU
            seuil_resol_implicite 1.e-5
    }
}
```

It defines the eight variables r_ω, r, T_u, T_l, H_u, H_l, L, K_ω and performs the execution of the command **flowrate -f**.

The output file `_output_flowrate_withKey_.dat` is with *"key=value"* type. It looks like:

```
yhat = 6.757218e+01;
d = 4.092561e+03;
```

yhat and **d** are defined as output variables in the macro.

XIV.4.12.2 Macro Uranie

```
void launchCodeFlowrateFlagSamplingKey(Int_t nS = 100)
{

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsFlowrate", "Design of experiments for Flowrate");

  // Add the study attributes ( min, max and nominal values)
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  // The reference input file
  TString sIn = TString("flowrate_input_with_flags.in");

  // Set the reference input file and the key for each input attributes
  tds->getAttribute("rw")->setFileKey(sIn, "@Rw@","%f",TAttributeFileKey::kFlag);
  tds->getAttribute("r")->setFileKey(sIn, "@R@","%f",TAttributeFileKey::kFlag);
  tds->getAttribute("tu")->setFileKey(sIn, "@Tu@","%f",TAttributeFileKey::kFlag);
  tds->getAttribute("tl")->setFileKey(sIn, "@Tl@","%f",TAttributeFileKey::kFlag);
  tds->getAttribute("hu")->setFileKey(sIn, "@Hu@","%f",TAttributeFileKey::kFlag);
  tds->getAttribute("hl")->setFileKey(sIn, "@Hl@","%f",TAttributeFileKey::kFlag);
  tds->getAttribute("l")->setFileKey(sIn, "@L@","%f",TAttributeFileKey::kFlag);
  tds->getAttribute("kw")->setFileKey(sIn, "@Kw@","%f",TAttributeFileKey::kFlag);

  // Generate the Design of Experiments
  TSampling *sampling = new TSampling(tds, "lhs", nS);
  sampling->generateSample();

  // The output file of the code
  TOutputFileKey *fout = new TOutputFileKey("_output_flowrate_withKey_.dat");
  // The attribute in the output file
  fout->addAttribute(new TAttribute("yhat"));
  fout->addAttribute(new TAttribute("d"));

  // Instanciation de mon code
  TCode *mycode = new TCode(tds, "flowrate -s -f ");
  // on ajoute le fichier de sortie du code
  mycode->addOutputFile( fout );

  // Lancement du code
  TLauncher *lanceur = new TLauncher(tds, mycode);
  lanceur->setSave();
  lanceur->setClean();
  //lanceur->setWorkingDirectory(gSystem->Getenv("PWD") + TString("/tmpLanceurUranie/ ↩
      flowrate"));
  lanceur->setVarDraw("yhat:rw","","");

  lanceur->run();



  tds->exportData("_flowrate_sampler_launcher_oat_.dat");

  // Visualisation
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro launchCodeFlowrateFlag" ↩
      ,5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  gStyle->SetPalette(1);
  pad->Divide(2, 2);
  pad->cd(1); tds->draw("yhat:rw");

  pad->cd(3); tds->draw("yhat");
  pad->cd(2); tds->draw("yhat:rw","","colz");

  pad->cd(4);
  // The parallel plot
  tds->draw("rw:r:tu:tl:hu:hl:l:kw:yhat","","para");
  TParallelCoord* para = (TParallelCoord*)gPad->GetListOfPrimitives()->FindObject(" ↩
      ParaCoord");

  // The output attribute
  TParallelCoordVar* axis = (TParallelCoordVar*)para->GetVarList()->FindObject("yhat");
  axis->AddRange(new TParallelCoordRange(axis,18.0,20.0));
  para->AddSelection("blue");
  para->GetCurrentSelection()->SetLineColor(kBlue);

  axis->AddRange(new TParallelCoordRange(axis,155.0,160.0));
  para->AddSelection("blue");
  para->GetCurrentSelection()->SetLineColor(kRed);

}
```

The distribution laws are set for each input variable. For example for the **rw** variable, the macro creates an uniform distribution between bounds 0.05 and 0.15 associated to a `TAttribute` named **"rw"**:

```
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
```

Then, the `TAttribute` is linked to the input file with:

```
tds->getAttribute("rw")->setFileFlag(sFileName, "@Rw@");
```

which is the same thing as writing:

```
tds->getAttribute("rw")->setFileKey("flowrate_input_with_flags.in", @Rw@,"%e", ↩
    TAttributeFileKey::kFlag);
```

which will be understood by Uranie as the fact the **rw** variable has to be read in an input file with flag format. The code will then, for each generated sampling, replace the flag @Rw@ in the input file by the corresponding value.

A design-of-experiments is then built with 100 samplings using the LHS method:

```
TSampling *sampling = new TSampling(tds, "lhs", 100);
sampling->generateSample();
```

The output file with "key=value" format `_output_flowrate_withKey_.dat` is instantiated as a `TOutputFileKey`.

```
TOutputFileKey *fout = new TOutputFileKey("_output_flowrate_withKey_.dat");
```

The two output variables **yhat** and **d** are then linked to this `TOutputFileKey` output file:

```
fout->addAttribute(new TAttribute("d"));
fout->addAttribute(new TAttribute("yhat"));
```

We set the code as being the **flowrate** execution with *"-f"* option:

```
TCode *mycode = new TCode(tds, "flowrate -f");
```

in which the "-f" option indicates that **flowrate** code has to find input files with "flags".

Then the launcher is initialised with a `TDataServer` and the code is executed:

```
TLauncher *tlch = new TLauncher(tds, mycode);
tlch->run();
```

The launcher will execute the **flowrate -f** command for each of the 100 patterns with the `run` method.

XIV.4.12.3 Graph

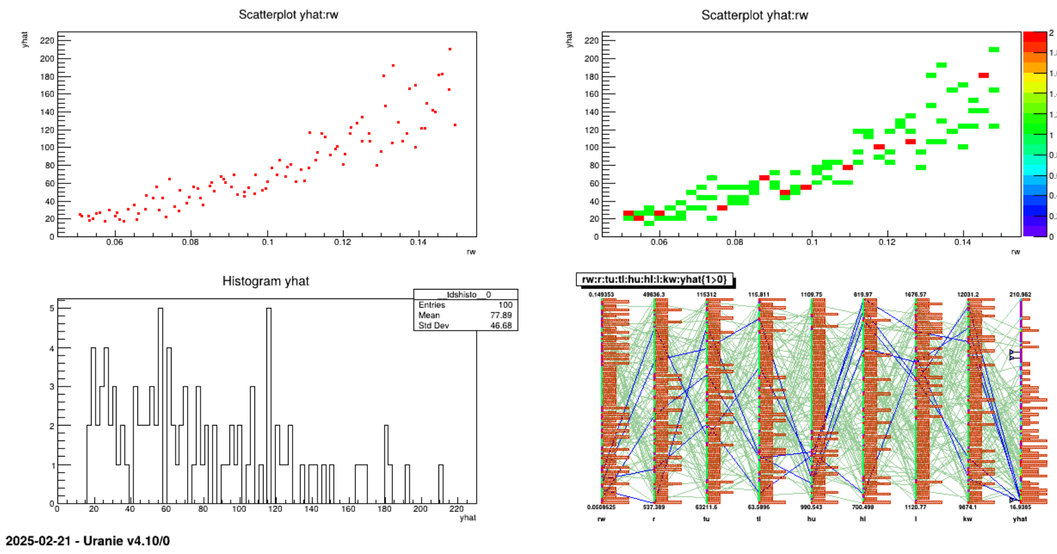


Figure XIV.28: Graph of the macro "launchCodeFlowrateFlagSamplingKey.C"

XIV.4.13 Macro "**launchCodeFlowrateKeyFlagSampling.C**"

XIV.4.13.1 Objective

The objective of the macro is to evaluate the **flowrate** external code on a design-of-experiments where the values of attributes will be replaced in two input files, the first, "flowrate_input_with_keys.in" with "key=value" type, the second, "flowrate_input_with_flags.in" with "flags" and the output file "_output_flowrate_withRow_.dat" with "values in rows" type. This **flowrate** code is described in Section IV.1.2.3.1.1.

The first input file flowrate_input_with_keys.in is "key=value" format:

```
#
#
# INPUT FILE with KEYS for the "FLOWREATE" code
# \date   2008-04-22 12:53:35
#


date = 123456 ;


#########################
##
##  exclude points
##
chu = 1050;
chl = 770;
cr  = 1100;
##
#########################

#########################
##
##  parameters : 8
```

```
##
Rw = 0.0500 ;
R = 33366.67 ;
Tu = 63070.0 ;
Tl = 116.00 ;
Hu = 1110.00 ;
Hl = 768.57;
L = 1200.0 ;
Kw = 11732.14 ;
##
#########################


#########################
##
## to simulate CPU time
##
## normal         1 :
## min     10000000 : 1.160u 0.000s 0:01.16 100.0%
## max    100000000 : 11.600u 0.010s 0:11.61 100.0%
##
nLoop = 1;
##
#########################

end = 6;
```

The creation of an input file containing some "flags" is motivated by the will to make evolve variables in an input file with no particular structure:

```
#
# INPUT FILE with FLAG for the "FLOWREATE" code
# \date   2008-04-22 12:55:17
#

new Implicit_Steady_State sch {
    frottement_paroi { 0.071623 19712.541454 }       // values of Rw and R
    tinit               0.0
    tmax                1000000.
    nb_pas_dt_max       1500
    dt_min              1051.972855                  // value of Hu
    dt_max              805.249178                   // value of Hl
    facsec              1000000.
    kW                  11401.611060                 // value of Kw
    information_Tu      Champ_Uniforme 1        85927.853162    // value of Tu
    information_Tl      Champ_Uniforme 1        85.803614       // value of Tl
    information_L {
        precision  1162.689830                       // value of L
    }
    convergence {
        criterion relative_max_du_dt
        precision  1.e-6
    }

    stop_criterium {
        ch_abcsissa_hu 1050
        ch_ordinate_hl 770
        c_radius 1100
    }

    Solveur Newton3 {
        max_iter_matrice        1
        max_iter_implicite      1
        date                    5654321
        seuil_convg_implicite   1.e-6
        assemblage_implicite    10
        solveur_lineaire        BiCGS
            preconditionneur    ILU
            seuil_resol_implicite 1.e-5
    }
}
```

These values are then replaced by "flags" bounded with special characters.

The second input file `flowrate_input_with_flags.in` containing "flags" is built this way:

```
#
# INPUT FILE with FLAG for the "FLOWREATE" code
# \date   2008-04-22 12:55:17
#

new Implicit_Steady_State sch {
    frottement_paroi { @Rw@ @R@ }
    tinit               0.0
    tmax                1000000.
    nb_pas_dt_max       1500
    dt_min              @Hu@
    dt_max              @Hl@
    facsec              1000000.
    kW                  @Kw@
    information_Tu      Champ_Uniforme 1         @Tu@
    information_Tl      Champ_Uniforme 1         @Tl@
    information_L {
        precision  @L@
    }
    convergence {
        criterion relative_max_du_dt
        precision  @Rw@
    }

    stop_criterium {
        ch_abcsissa_hu 1050
        ch_ordinate_hl 770
        c_radius 1100
    }

    Solveur Newton3 {
        max_iter_matrice        1
        max_iter_implicite      1
        date                    5654321
        seuil_convg_implicite   1.e-6
        assemblage_implicite    10
        solveur_lineaire        BiCGS
            preconditionneur    ILU
            seuil_resol_implicite 1.e-5
    }
}
```

These files define the eight variables r_ω, r, T_u, T_l, H_u, H_l, L, K_ω needed to perform the execution of the command **flowrate -k** and the command **flowrate -f**.

The output file `_output_flowrate_withRow_.dat` is of "values in rows" type. It looks like:

```
#COLUMN_NAMES: yhat | d
```

```
6.757218e+01  4.092561e+03
```

yhat and **d** are defined as output variables in the macro.

XIV.4.13.2 Macro Uranie

```
void launchCodeFlowrateKeyFlagSampling(Int_t nS = 100)
{

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsFlowrate", "Design of experiments for Flowrate");

  // Add the study attributes ( min, max and nominal values)
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  // The reference input file with key
  TString sInKey = TString("flowrate_input_with_keys.in");

  // Set the reference input file and the key for each input attributes
  tds->getAttribute("rw")->setFileKey(sInKey, "Rw");
  tds->getAttribute("r")->setFileKey(sInKey, "R");
  tds->getAttribute("tu")->setFileKey(sInKey, "Tu");
  tds->getAttribute("tl")->setFileKey(sInKey, "Tl");
  tds->getAttribute("hu")->setFileKey(sInKey, "Hu");
  tds->getAttribute("hl")->setFileKey(sInKey, "Hl");
  tds->getAttribute("l")->setFileKey(sInKey, "L");
  tds->getAttribute("kw")->setFileKey(sInKey, "Kw");

  // The reference input file with flag
  TString sInFlag = TString("flowrate_input_with_flags.in");
  tds->getAttribute("rw")->setFileFlag(sInFlag, "@Rw@");
  tds->getAttribute("r")->setFileFlag(sInFlag, "@R@");
  tds->getAttribute("tu")->setFileFlag(sInFlag, "@Tu@");
  tds->getAttribute("tl")->setFileFlag(sInFlag, "@Tl@");
  tds->getAttribute("hu")->setFileFlag(sInFlag, "@Hu@");
  tds->getAttribute("hl")->setFileFlag(sInFlag, "@Hl@");
  tds->getAttribute("l")->setFileFlag(sInFlag, "@L@");
  tds->getAttribute("kw")->setFileFlag(sInFlag, "@Kw@");

  // Generate the Design of Experiments
  TSampling *sampling = new TSampling(tds, "lhs", nS);
  sampling->generateSample();

  // The output file of the code
  TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
  // The attribute in the output file
  fout->addAttribute(new TAttribute("yhat"));

  // Instanciation de mon code
  TCode *mycode = new TCode(tds, "flowrate -s -k");
  // on ajoute le fichier de sortie du code
  mycode->addOutputFile( fout );
  // Lancement du code
  TLauncher *lanceur = new TLauncher(tds, mycode);
  lanceur->setSave();
  lanceur->setClean();
  //lanceur->setWorkingDirectory(gSystem->Getenv("PWD") + TString("/tmpLanceurUranie/ ↩
      flowrate"));
  lanceur->setVarDraw("yhat:rw","","");

  lanceur->run();



//   tds->scan("*");
  tds->exportData("_flowrate_sampler_launcher_.dat");

   // Visualisation
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro launchCodeFlowrateFlag" ↩
      ,5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  gStyle->SetPalette(1);
  pad->Divide(2, 2);
  pad->cd(1); tds->draw("yhat:rw");

  pad->cd(3); tds->draw("yhat");
  pad->cd(2); tds->draw("yhat:rw","","colz");

  pad->cd(4);
  // The parallel plot
  tds->draw("rw:r:tu:tl:hu:hl:l:kw:yhat","","para");
  TParallelCoord* para = (TParallelCoord*)gPad->GetListOfPrimitives()->FindObject(" ↩
      ParaCoord");

  // The output attribute
  TParallelCoordVar* axis = (TParallelCoordVar*)para->GetVarList()->FindObject("yhat");
  axis->AddRange(new TParallelCoordRange(axis,15.0,80.0));
  para->AddSelection("blue");
  para->GetCurrentSelection()->SetLineColor(kBlue);

}
```

The laws of distribution are set for each input variable. For example for the **rw** variable, the macro creates an uniform distribution between bounds 0.05 and 0.15 associated to a `TAttribute` named **"rw"**:

```
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
```

Then, the `TAttribute` is linked to the first input file with:

```
tds->getAttribute("rw")->setFileKey(TString("flowrate_input_with_keys.in"), "Rw");
```

which will be understood by Uranie as the fact the **rw** variable has to be read in an input file with "key=value" format. If the user wants to get values of variables in an input file with "flags", he can redefine the link towards the "flag" file directly:

```
tds->getAttribute("rw")->setFileFlag(TString("flowrate_input_with_flags.in"), "@Rw@");
```

So the code will look for input files with "flags".

A design-of-experiments is then built with 100 samplings using the LHS method:

```
TSampling *sampling = new TSampling(tds, "lhs", 100);
sampling->generateSample();
```

The output file with "values in rows" type `_output_flowrate_withRow_.dat` is instantiated as a `TOutputFileRow`

```
TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
```

The variables **yhat** and **d** could then be linked to this `TOutputFileRow` output file. But here, only **yhat** will be considered as a variable of interest by Uranie.

```
fout->addAttribute("yhat");
```

We set the code as being the **flowrate** execution with *"-f"* option but user can also use the *"-k"* option.

```
TCode *mycode = new TCode(tds, "flowrate -f");
```

which indicates that **flowrate** code has to find input files with "flags". With option *"-k"*, **flowrate** has to find file with "key=value" format.

Then the launcher is initialised with a `TDataServer` and the code is executed:

```
TLauncher *tlch = new TLauncher(tds, mycode);
```

The launcher will execute the **flowrate -f** command for each of the 100 patterns with the `run` method:

```
tlch->run();
```

XIV.4.13.3 Graph

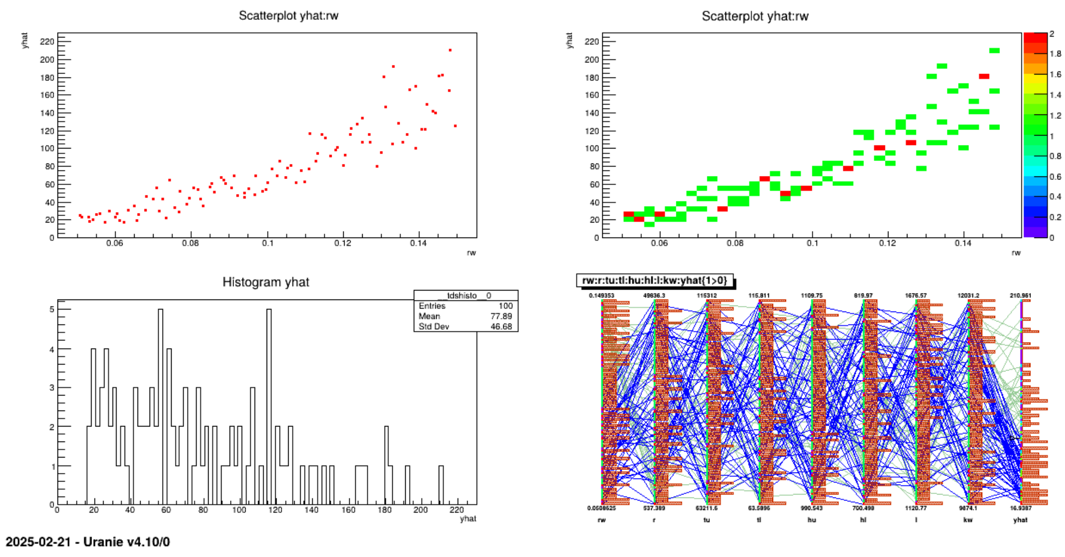


Figure XIV.29: Graph of the macro "`launchCodeFlowrateKeyFlagSampling.C`"

XIV.4.14 Macro "launchCodeFlowrateKeywithFlagSampling.C"

XIV.4.14.1 Objective

The objective of the macro is to evaluate the **flowrate** external code on a design-of-experiments where the input file "`flowrate_input_with_keys.in`" is with "key=value" type and defines the variables usable by the code ; a second input file "`flowrate_input_with_flags.in`" will moved by Uranie in the working directory but it will not be modified. The output file "`_output_flowrate_withRow_.dat`" is with "values in rows" type. This code is described in Section IV.1.2.3.1.1.

The first input file `flowrate_input_with_keys.in` has "key=value" format:

```
#
#
# INPUT FILE with KEYS for the "FLOWREATE" code
# \date   2008-04-22 12:53:35
#


date = 123456 ;


#########################
##
##  exclude points
##
chu = 1050;
chl = 770;
cr  = 1100;
##
#########################

#########################
##
##  parameters : 8
##
Rw = 0.0500 ;
R = 33366.67 ;
Tu = 63070.0 ;
Tl = 116.00 ;
Hu = 1110.00 ;
Hl = 768.57;
L = 1200.0 ;
Kw = 11732.14 ;
##
#########################


#########################
##
## to simulate CPU time
##
## normal         1 :
## min     10000000 : 1.160u 0.000s 0:01.16 100.0%
## max    100000000 : 11.600u 0.010s 0:11.61 100.0%
##
nLoop = 1;
##
#########################

end = 6;
```

If we suppose that one wants to add an input file with "flag" format `flowrate_input_with_flags.in`.

```
#
# INPUT FILE with FLAG for the "FLOWREATE" code
# \date   2008-04-22 12:55:17
#
```

```
new Implicit_Steady_State sch {
    frottement_paroi { @Rw@ @R@ }
    tinit                0.0
    tmax                 1000000.
    nb_pas_dt_max        1500
    dt_min               @Hu@
    dt_max               @Hl@
    facsec               1000000.
    kW                   @Kw@
    information_Tu       Champ_Uniforme 1         @Tu@
    information_Tl       Champ_Uniforme 1         @Tl@
    information_L {
        precision   @L@
    }
    convergence {
        criterion relative_max_du_dt
        precision   @Rw@
    }

    stop_criterium {
        ch_abcsissa_hu 1050
        ch_ordinate_hl 770
        c_radius 1100
    }

    Solveur Newton3 {
        max_iter_matrice        1
        max_iter_implicite      1
        date                    5654321
        seuil_convg_implicite   1.e-6
        assemblage_implicite    10
        solveur_lineaire        BiCGS
            preconditionneur    ILU
            seuil_resol_implicite 1.e-5
    }
}
```

The input file `flowrate_input_with_keys.in` defines the eight variables r_ω, r, T_u, T_l, H_u, H_l, L, K_ω needed to perform the execution of the command **flowrate -k**.

The output file `_output_flowrate_withRow_.dat` is with "values in rows" type. It looks like:

```
#COLUMN_NAMES: yhat | d

6.757218e+01  4.092561e+03
```

yhat and **d** have to be defined as output variables in the macro. But in this one, only **yhat** will be considered.

XIV.4.14.2 Macro Uranie

```
void launchCodeFlowrateKeywithFlagSampling(Int_t nS = 100)
{

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsFlowrate", "Design of experiments for Flowrate");

  // Add the study attributes ( min, max and nominal values)
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  // The reference input file with key
  TString sInKey = TString("flowrate_input_with_keys.in");

  // Set the reference input file and the key for each input attributes
  tds->getAttribute("rw")->setFileKey(sInKey, "Rw");
  tds->getAttribute("r")->setFileKey(sInKey, "R");
  tds->getAttribute("tu")->setFileKey(sInKey, "Tu");
  tds->getAttribute("tl")->setFileKey(sInKey, "Tl");
  tds->getAttribute("hu")->setFileKey(sInKey, "Hu");
  tds->getAttribute("hl")->setFileKey(sInKey, "Hl");
  tds->getAttribute("l")->setFileKey(sInKey, "L");
  tds->getAttribute("kw")->setFileKey(sInKey, "Kw");

  // Generate the Design of Experiments
  TSampling *sampling = new TSampling(tds, "lhs", nS);
  sampling->generateSample();

  // The output file of the code
  TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
  // The attribute in the output file
  fout->addAttribute(new TAttribute("yhat"));

  // Instanciation de mon code
  TCode *mycode = new TCode(tds, "flowrate -s -k");
  // on ajoute le fichier de sortie du code
  mycode->setReferenceDirectory( gSystem->pwd() );
  mycode->addInputFile("flowrate_input_with_flags.in");

  mycode->addOutputFile( fout );

  // Lancement du code
  TLauncher *lanceur = new TLauncher(tds, mycode);
  //lanceur->setWorkingDirectory(gSystem->Getenv("PWD") + TString("/tmpLanceurUranie/ ↩
      flowrate"));
  lanceur->setVarDraw("yhat:rw","","");

  lanceur->run();

  // Visualisation
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro launchCodeFlowrateOATMinMax" ↩
      ,5,64,1270,667);
  gStyle->SetPalette(1);
  tds->draw("yhat:rw");



  tds->startViewer();
}
```

The laws of distribution are set for each input variable. For example for the **rw** variable, the macro creates an uniform distribution between bounds 0.05 and 0.15 associated to a `TAttribute` named **"rw"**:

```
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
```

Then, the `TAttribute` is linked to the input file with:

```
tds->getAttribute("rw")->setFileKey(TString("flowrate_input_with_keys.in"), "Rw");
```

which will be understood by Uranie as the fact the **rw** variable has to be read in an input file with "key=value" format.

A design-of-experiments is then built with 100 samplings using the LHS method.

```
TSampling *sampling = new TSampling(tds, "lhs", 100);
sampling->generateSample();
```

The output file with "values in rows" type `_output_flowrate_withKey_.dat` is set as a `TOutputFileRow`.

```
TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
```

The variable **yhat** is then linked to this `TOutputFileRow` output file.

```
fout->addAttribute(new TAttribute("yhat"));
```

We set the code as being the **flowrate** execution with *"-k"* option:

```
TCode *mycode = new TCode(tds, "flowrate -k");
```

in which the "-k" option indicates that **flowrate** code has to find "key=value" input files.

We suppose the user wants to add another input file to the code. He specifies where is the new input file:

```
mycode->setReferenceDirectory( gSystem->pwd() );
```

and then he adds the new input file (here it's an input file with "flag" format) which will be copied each time in the working directory.

```
mycode->addInputFile("flowrate_input_with_flags.in");
```

Then the launcher is initialised with a `TDataServer` and the code is executed and launched with the `run` method

```
TLauncher *lanceur = new TLauncher(tds, mycode);
lanceur->run();
```

XIV.4.14.3 Graph

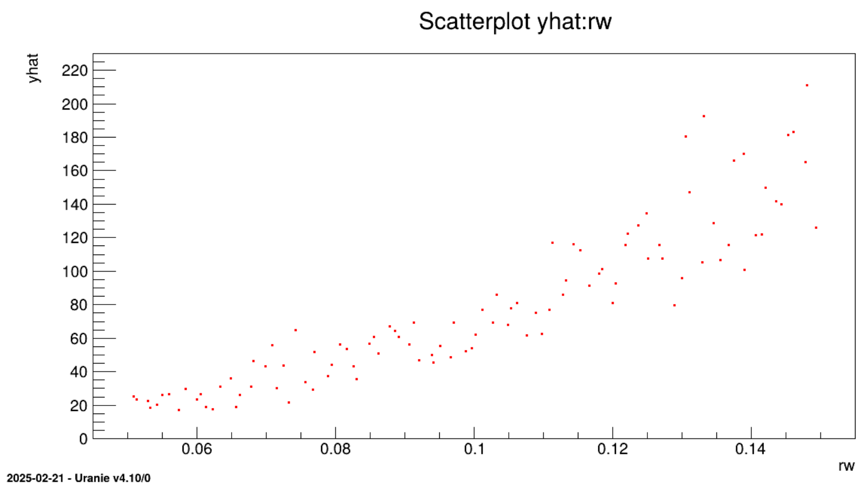


Figure XIV.30: Graph of the macro "`launchCodeFlowrateKeywithFlagSampling.C`"

XIV.4.15 Macro "launchCodeFlowrateKeyFailure.C"

XIV.4.15.1 Objective

The objective of the macro is to evaluate the external code flowrate with failures on a design-of-experiments where input file "flowrate_input_with_keys.in" is with "key=value" type and the output file "_output_flowrate_withRow_.dat" with type "values in rows". This **flowrate** code is described in Section IV.1.2.3.1.1.

The input file flowrate_input_with_keys.in is with "key=value":

```
#
#
# INPUT FILE with KEYS for the "FLOWREATE" code
# \date   2008-04-22 12:53:35
#


date = 123456 ;


#########################
##
##  exclude points
##
chu = 1050;
chl = 770;
cr  = 1100;
##
#########################

#########################
##
##  parameters : 8
##
Rw = 0.0500 ;
R = 33366.67 ;
Tu = 63070.0 ;
Tl = 116.00 ;
Hu = 1110.00 ;
Hl = 768.57;
L = 1200.0 ;
Kw = 11732.14 ;
##
#########################


#########################
##
## to simulate CPU time
##
## normal         1 :
## min     10000000 : 1.160u 0.000s 0:01.16 100.0%
## max    100000000 : 11.600u 0.010s 0:11.61 100.0%
##
nLoop = 1;
##
#########################

end = 6;
```

This file defines the eight variables r_ω, r, T_u, T_l, H_u, H_l, L, K_ω and also the three additional variables **chu**, **chl** and **cr** needed to perform the execution of the command **flowrate -kf**.

The output file `_output_flowrate_withRow_.dat` is of "values in rows" type. It looks like:

```
#COLUMN_NAMES: yhat | d

6.757218e+01  4.092561e+03
```

yhat and **d** are defined as output variables in the macro.

XIV.4.15.2 Macro Uranie

```
{

  Int_t nS = 400;
  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsflowrate", "DataBase flowrate");
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  // The reference input file
  TString sIn = gSystem->pwd() + TString("/");
  sIn += TString("flowrate_input_with_keys.in");
  tds->getAttribute("rw")->setFileKey(sIn, "Rw");
  tds->getAttribute("r")->setFileKey(sIn, "R");
  tds->getAttribute("tu")->setFileKey(sIn, "Tu");
  tds->getAttribute("tl")->setFileKey(sIn, "Tl");
  tds->getAttribute("hu")->setFileKey(sIn, "Hu");
  tds->getAttribute("hl")->setFileKey(sIn, "Hl");
  tds->getAttribute("l")->setFileKey(sIn, "L");
  tds->getAttribute("kw")->setFileKey(sIn, "Kw");

  // Generate the sample
  TSampling *sampling = new TSampling(tds, "lhs", nS);
  sampling->generateSample();

  // The output file of the code
  TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
  // The attribute in the output file
  TAttribute * tyhat = new TAttribute("yhat");
  tyhat->setDefaultValue(-200.0);
  fout->addAttribute(tyhat);

  // Create a TCode object
  TCode *mycode = new TCode(tds, "flowrate -s -kf");
  // Add the output file where to find the output attributes
  //  mycode->addInputFile(new TInputFile(sIn));
  mycode->addOutputFile( fout );

  // Create a TLauncher object from a TDataServer and a TCode objects
  TLauncher *lanceur = new TLauncher(tds, mycode);
  lanceur->setSave();
  // lanceur->setClean();
  lanceur->setVarDraw("hu:hl","yhat>0","");
  // lanceur->setWorkingDirectory(gSystem->Getenv("PWD") + TString("/tmpLanceurUranie/ ↩
      flowrate"));

  lanceur->run();


   // Visualisation
   TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro launchCodeFlowratestop" ↩
       ,5,64,1270,667);
   tds->draw("hu:hl","yhat>0");



  tds->exportData("_flowrate_sampler_launcher_.dat");
}
```

The laws of distribution are set for each input variable. For example for the **rw** variable, the macro creates an uniform distribution between bounds 0.05 and 0.15 associated to a `TAttribute` named **"rw"**:

```
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
```

Then, the `TAttribute` is linked to the input file with:

```
tds->getAttribute("rw")->setFileKey(TString("flowrate_input_with_keys.in"), "Rw");
```

which will be understood by Uranie as the fact the **rw** variable has to be read in an input file with "key=value" format.

A design-of-experiments is then built with 300 samplings using the LHS method.

```
TSampling *sampling = new TSampling(tds, "lhs", 300);
sampling->generateSample();
```

The output file with "values in rows" `_output_flowrate_withRow_.dat` is instantiated as a `TOutputFileRow`.

```
TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
```

The variable **yhat** and **d** are then linked to this `TOutputFileRow` output file. But here, only **yhat** will be considered as variable of interest by Uranie.

```
fout->addAttribute("yhat");
```

We set the code as being the **flowrate** execution with *"-kf"* option.

```
TCode *mycode = new TCode(tds, "flowrate -v -kf");
```

which indicates that **flowrate** code has to find input files with "key=value" format. Furthermore, the additional variables **chu**, **chl** and **cr** allow to check if *flowrate* is in failure or not. If it is in failure, the output file will not be produced.

Then the launcher is initialised with a `TDataServer` and the code is executed.

```
TLauncher *lanceur = new TLauncher(tds, mycode);
```

The launcher will execute the **flowrate -kf** command for each of the 300 patterns with the `run` method.

```
lanceur->run();
```

XIV.4.15.3 Graph

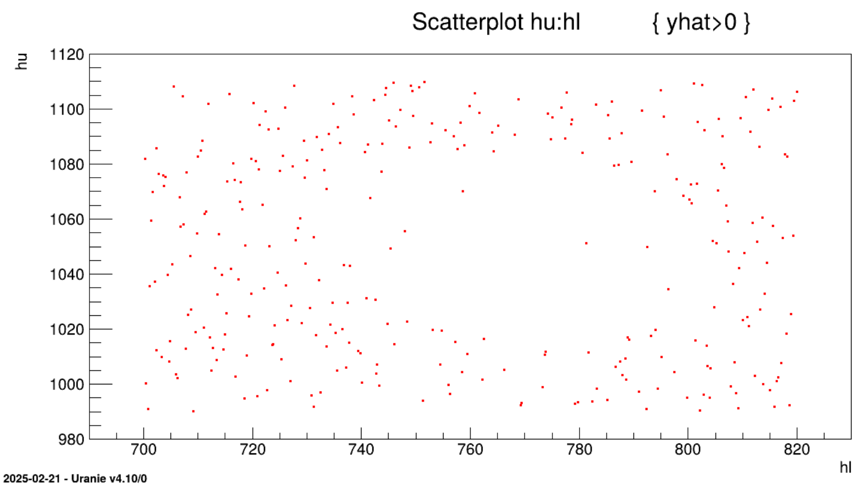


Figure XIV.31: Graph of the macro "launchCodeFlowrateKeyFailure.C"

XIV.4.15.4 Console

The macro simulates 100 failures of **flowrate**.

```
--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 <URANIE::INFO>
 <URANIE::INFO> TLauncher "0x5a749de41ef0" TCode "codetdsflowrate" with TDataServer " ↩
    tdsflowrate"
 <URANIE::INFO>  ** Save : Yes
 <URANIE::INFO>  ** WorkingDirectory[${RUNNINGDIR}/URANIE]
 <URANIE::INFO>  ** Proc : 1
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
 <URANIE::INFO>
 <URANIE::INFO> time elapsed 3.04 sec
 <URANIE::INFO>      Failure : 92/400 --- 23%
```

XIV.4.16 Macro "`launchCodeFlowrateFlagFailure.C`"

XIV.4.16.1 Objective

The objective of the macro is to evaluate the external code flowrate with failures on a design-of-experiments where input file "`flowrate_input_with_flags.in`" has"flags" and the output file "`_output_flowrate_withRow_.dat`" with type "values in rows". This **flowrate** code is described in Section IV.1.2.3.1.1.

The creation of an input file containing some "flags" is motivated by the will to make evolve variables in an input file with no particular structure:

```
#
# INPUT FILE with FLAG for the "FLOWREATE" code
# \date   2008-04-22 12:55:17
#

new Implicit_Steady_State sch {
    frottement_paroi { 0.071623 19712.541454 }       // values of Rw and R
    tinit               0.0
    tmax                1000000.
    nb_pas_dt_max       1500
    dt_min              1051.972855                  // value of Hu
    dt_max              805.249178                   // value of Hl
    facsec              1000000.
    kW                  11401.611060                 // value of Kw
    information_Tu      Champ_Uniforme 1        85927.853162    // value of Tu
    information_Tl      Champ_Uniforme 1        85.803614       // value of Tl
    information_L {
        precision  1162.689830                       // value of L
    }
    convergence {
        criterion relative_max_du_dt
        precision  1.e-6
    }

    stop_criterium {
        ch_abcsissa_hu 1050
        ch_ordinate_hl 770
        c_radius 1100
    }

    Solveur Newton3 {
        max_iter_matrice        1
        max_iter_implicite      1
        date                    5654321
        seuil_convg_implicite   1.e-6
        assemblage_implicite    10
        solveur_lineaire        BiCGS
            preconditionneur    ILU
            seuil_resol_implicite 1.e-5
    }
}
```

These values are then replaced by "flags" bounded with special characters.

The input file `flowrate_input_with_flags.in` containing "flags" is built this way:

```
#
# INPUT FILE with FLAG for the "FLOWREATE" code
# \date   2008-04-22 12:55:17
#

new Implicit_Steady_State sch {
    frottement_paroi { @Rw@ @R@ }
    tinit               0.0
    tmax                1000000.
    nb_pas_dt_max       1500
    dt_min              @Hu@
    dt_max              @Hl@
    facsec              1000000.
    kW                  @Kw@
    information_Tu      Champ_Uniforme 1        @Tu@
    information_Tl      Champ_Uniforme 1        @Tl@
    information_L {
        precision  @L@
    }
    convergence {
        criterion relative_max_du_dt
        precision  @Rw@
    }

    stop_criterium {
        ch_abcsissa_hu 1050
        ch_ordinate_hl 770
        c_radius 1100
    }

    Solveur Newton3 {
        max_iter_matrice        1
        max_iter_implicite      1
        date                    5654321
        seuil_convg_implicite   1.e-6
        assemblage_implicite    10
        solveur_lineaire        BiCGS
            preconditionneur    ILU
            seuil_resol_implicite 1.e-5
    }
}
```

This file defines the eight variables r_ω, r, T_u, T_l, H_u, H_l, L, K_ω and also the three additional variables **chu**, **chl** and **cr** needed to perform the execution of the command **flowrate -kf**.

The output file _output_flowrate_withRow_.dat is of "values in rows" type. It looks like:

```
#COLUMN_NAMES: yhat | d

6.757218e+01  4.092561e+03
```

yhat and **d** are defined as output variables in the macro.

XIV.4.16.2 Macro Uranie

```
{

  Int_t nS = 300;
  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsflowrate", "DataBase flowrate");
  // Add the eight attributes of the study with uniform law
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  // The input file
  TString sFileName = TString("flowrate_input_with_flags.in");

  // Set the input file with flag format and the flag for each input attributes
  tds->getAttribute("rw")->setFileFlag(sFileName, "@Rw@");
  tds->getAttribute("r")->setFileFlag(sFileName, "@R@");
  tds->getAttribute("tu")->setFileFlag(sFileName, "@Tu@");
  tds->getAttribute("tl")->setFileFlag(sFileName, "@Tl@");
  tds->getAttribute("hu")->setFileFlag(sFileName, "@Hu@");
  tds->getAttribute("hl")->setFileFlag(sFileName, "@Hl@");
  tds->getAttribute("l")->setFileFlag(sFileName, "@L@");
  tds->getAttribute("kw")->setFileFlag(sFileName, "@Kw@");


  // Generate the sample
  TSampling *sampling = new TSampling(tds, "lhs", 400);
  sampling->generateSample();

  // The output file of the code where values are stored in row
  TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
  // The attribute in the output file
  TAttribute * tyhat = new TAttribute("yhat");
  tyhat->setDefaultValue(-200.0);
  fout->addAttribute(tyhat);

  // Create a TCode object
  TCode *mycode = new TCode(tds, "flowrate -s -ff");
  // Add the output file where to find the output attributes
  mycode->addOutputFile( fout );

  // Create a TLauncher object from a TDataServer and a TCode objects
  TLauncher *tlch = new TLauncher(tds, mycode);
  tlch->setSave();
  // tlch->setClean();
  tlch->setVarDraw("hu:hl","yhat>0","");
  // tlch->setWorkingDirectory(gSystem->Getenv("PWD") + TString("/tmpLanceurUranie/flowrate ↩
      "));

  tlch->run();

  // Visualisation
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro launchCodeFlowrateFlagFailure" ↩
      ,5,64,1270,667);
  tds->draw("hu:hl","yhat>0","");


  tds->exportData("_flowrate_sampler_launcher_.dat");
}
```

The laws of distribution are set for each input variable. For example for the **rw** variable, the macro creates an uniform distribution between bounds 0.05 and 0.15 associated to a `TAttribute` named **"rw"**:

```
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
```

Then, the `TAttribute` is linked to the input file with:

```
tds->getAttribute("rw")->setFileFlag(TString("flowrate_input_with_flags.in"),"@Rw@");
```

which will be understood by Uranie as the fact the **rw** variable has to be read in an input file with "flags".

A design-of-experiments is then built with 300 samplings using the LHS method.

```
TSampling *sampling = new TSampling(tds, "lhs", 300);
sampling->generateSample();
```

The output file with "values in rows" type `_output_flowrate_withRow_.dat` is instantiated as a `TOutputFileRow`.

```
TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
```

The variable **yhat** and **d** are then linked to this `TOutputFileRow` output file. But here, only **yhat** will be considered as a variable of interest by Uranie.

```
fout->addAttribute("yhat");
```

We set the code as being the **flowrate** execution with *"-ff"* option.

```
TCode *mycode = new TCode(tds, "flowrate -ff");
```

which indicates that **flowrate** code has to find input files with "flags" format. Furthermore, the additional variables **chu**, **chl** and **cr** allow to check if *flowrate* is in failure or not. If it is in failure, the output file will not be produced.

Then the launcher is initialised with a `TDataServer` and the code is executed.

```
TLauncher *tlch = new TLauncher(tds, mycode);
```

The launcher will execute the **flowrate -ff** command for each of the 300 patterns with the `run` method.

```
tlch->run();
```

XIV.4.16.3 Graph

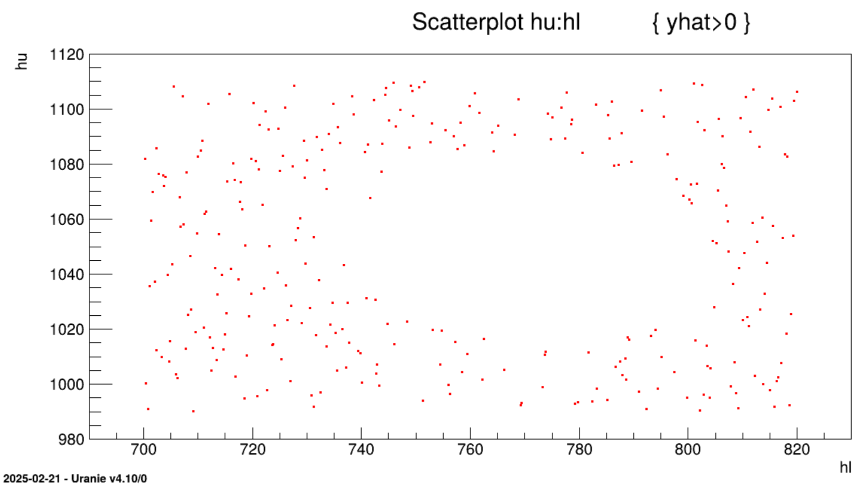


Figure XIV.32: Graph of the macro "launchCodeFlowrateFlagFailure.C"

XIV.4.16.4 Console

The macro simulates 100 failures for **flowrate**.

```
--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 <URANIE::INFO>
 <URANIE::INFO> TLauncher "0x601ca21ab960" TCode "codetdsflowrate" with TDataServer " ↩
     tdsflowrate"
 <URANIE::INFO>  ** Save : Yes
 <URANIE::INFO>  ** WorkingDirectory[${RUNNINGDIR}/URANIE]
 <URANIE::INFO>  ** Proc : 1
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
job exit code is out
 <URANIE::INFO>
 <URANIE::INFO> time elapsed 3.08 sec
 <URANIE::INFO>      Failure : 100/400 --- 25%
```

XIV.4.17 Macro "launchCodeFlowrateKeyOATMinMax.C"

XIV.4.17.1 Objective

The objective of the macro is to evaluate the **flowrate** external code on a design-of-experiments generated by the function PlanOATMinMax where input file "flowrate_input_with_keys.in" is with "key=value" format and the output file "_output_flowrate_withRow_.dat" is with "values in rows" type. This code is described in Section IV.1.2.1. The function PlanOATMinMax consists in constructing a design around the nominal value for each attribute, except for one attribute whose value is set to one of its boundaries(minimum and then maximum). Doing this for the 8 input attributes and the central design, in which each attribute takes its nominal value leads to a seventeen patterns design.

The input file for this code is flowrate_input_with_keys.in of *key = value* type. It looks like:

```
#
#
# INPUT FILE with KEYS for the "FLOWREATE" code
# \date   2008-04-22 12:53:35
#


date = 123456 ;


#########################
##
##  exclude points
##
chu = 1050;
chl = 770;
cr  = 1100;
##
#########################

#########################
##
##  parameters : 8
##
```

```
Rw = 0.0500 ;
R = 33366.67 ;
Tu = 63070.0 ;
Tl = 116.00 ;
Hu = 1110.00 ;
Hl = 768.57;
L = 1200.0 ;
Kw = 11732.14 ;
##
#########################


#########################
##
## to simulate CPU time
##
## normal         1 :
## min     10000000 : 1.160u 0.000s 0:01.16 100.0%
## max    100000000 : 11.600u 0.010s 0:11.61 100.0%
##
nLoop = 1;
##
#########################

end = 6;
```

These files define the eight variables r_ω, r, T_u, T_l, H_u, H_l, L, K_ω needed to perform the execution of the command **flowrate -k**.

The output file `_output_flowrate_withRow_.dat` is of "values in rows" type. It looks like:

```
#COLUMN_NAMES: yhat | d

6.757218e+01  4.092561e+03
```

yhat and **d** are defined as output variables in the macro.

XIV.4.17.2 Macro Uranie

```
void PlanOATMinMax(TDataServer *tds)
{
  // Create the data tuple
  tds->createTuple();

  Int_t natt = tds->getNAttributes();
  // The number of attributes is incremented by one for the iterator
  const Int_t nn = natt+1;

  // Allocate the data vector to fill the tuple
  Double_t value[nn];

  // Init the iterator. It is always the first value.
  value[0] = 0;
  // Init the data vector with the nominal/mean values of attributes
  for(Int_t i=0;i<natt;i++)
    value[1+i] = tds->getAttribute(i)->getMean();

  // Fill the nominal/mean values
  tds->getTuple()->Fill(value);
  // Loop on each attributes
  for(Int_t i=0;i<natt;i++) {
    // Case of min values
    value[1+i] = tds->getAttribute(i)->getLowerBound();
    value[0] += 1.0;
    tds->getTuple()->Fill(value);
    // Case of the max value
    value[1+i] = tds->getAttribute(i)->getUpperBound();
    value[0] += 1.0;
    tds->getTuple()->Fill(value);
    // Reset the nominal/mean value
    value[1+i] = tds->getAttribute(i)->getMean();
  }
}

void launchCodeFlowrateKeyOATMinMax()
{

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsOATMinMaxFlowrate", "OATMinMax Design of ↩
      experiments for Flowrate");

  // Add the study attributes ( min, max and nominal values)
  tds->addAttribute( new TAttribute("rw", 0.05, 0.15));
  tds->getAttribute("rw")->setMean(0.10);

  tds->addAttribute( new TAttribute("r", 100.0, 50000.0));
  tds->getAttribute("r")->setMean(25050.);

  tds->addAttribute( new TAttribute("tu", 63070.0, 115600.0));
  tds->getAttribute("tu")->setMean(89335.);

  tds->addAttribute( new TAttribute("tl", 63.1, 116.0));
  tds->getAttribute("tl")->setMean(89.55);

  tds->addAttribute( new TAttribute("hu", 990.0, 1110.0));
  tds->getAttribute("hu")->setMean(1050.);

  tds->addAttribute( new TAttribute("hl", 700.0, 820.0));
  tds->getAttribute("hl")->setMean(760.);

  tds->addAttribute( new TAttribute("l", 1120.0, 1680.0));
  tds->getAttribute("l")->setMean(1400.);

  tds->addAttribute( new TAttribute("kw", 9855.0, 12045.0));
  tds->getAttribute("kw")->setMean(10950.);

  // The reference input file
  TString sIn = gSystem->pwd();
  sIn += TString("/flowrate_input_with_keys.in");

  // Set the reference input file and the key for each input attributes
  tds->getAttribute("rw")->setFileKey(sIn, "Rw");
  tds->getAttribute("r")->setFileKey(sIn, "R");
  tds->getAttribute("tu")->setFileKey(sIn, "Tu");
  tds->getAttribute("tl")->setFileKey(sIn, "Tl");
  tds->getAttribute("hu")->setFileKey(sIn, "Hu");
  tds->getAttribute("hl")->setFileKey(sIn, "Hl");
  tds->getAttribute("l")->setFileKey(sIn, "L");
  tds->getAttribute("kw")->setFileKey(sIn, "Kw");

  // Construit le plan OAT par avec les bornes (Min, Max) and the nominal values
```

```
   PlanOATMinMax(tds);

  // Scan the data ( 17 = 1 + 2*8 patterns)
  tds->Scan("*","","colsize=5 col=3:4::6::4:3:4:");

  // Save the Design of experiments in an ASCII file
  tds->exportData("_flowrate_sampler_oat_.dat");

  // The output file of the code
  TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
  // The attribute in the output file
  fout->addAttribute(new TAttribute("yhat"));

  // Instanciation de mon code
  TCode *mycode = new TCode(tds, "flowrate -s -k");
  // mycode->addInputFile( new TInputFile(sIn));
  // on ajoute le fichier de sortie du code
  mycode->addOutputFile( fout );

  // Lancement du code
  TLauncher *lanceur = new TLauncher(tds, mycode);
  lanceur->setSave();
  lanceur->setClean();
  //lanceur->setWorkingDirectory(gSystem->Getenv("PWD") + TString("/tmpLanceurUranie/ ↩
      flowrate"));
  lanceur->setVarDraw("yhat:rw","","");

  lanceur->run();



  tds->exportData("_flowrate_sampler_launcher_oat_.dat");

  // Visualisation
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro launchCodeFlowrateOATMinMax" ↩
      ,5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  gStyle->SetPalette(1);
  pad->Divide(2, 2);
  pad->cd(1); tds->draw("yhat:rw");

  pad->cd(3); tds->draw("yhat");
  pad->cd(2); tds->draw("yhat:rw","","colz");

  pad->cd(4);
  // The parallel plot
  tds->draw("rw:r:tu:tl:hu:hl:l:kw:yhat","","para");
  TParallelCoord* para = (TParallelCoord*)gPad->GetListOfPrimitives()->FindObject(" ↩
      ParaCoord");

  // The output attribute
  TParallelCoordVar* axis = (TParallelCoordVar*)para->GetVarList()->FindObject("yhat");
  axis->AddRange(new TParallelCoordRange(axis,15.0,20.0));
  para->AddSelection("blue");
  para->GetCurrentSelection()->SetLineColor(kBlue);

  axis->AddRange(new TParallelCoordRange(axis,155.0,160.0));
  para->AddSelection("blue");
  para->GetCurrentSelection()->SetLineColor(kBlue);

}
```

PlanOATMinMax function builds the design-of-experiments from a TDataServer: for the first design and for each attribute of the TDataServer, values of attributes are initialised with their means.

```
for(Int_t i=0 ; i<tds->getNAttributes(); i++)
value[1+i] = tds->getAttribute(i)->getMean();
```

These data are put in the TDataServer,

```
tds->getTuple()->Fill(value)
```

this is how the central point is constructed. Two designs are then built for every attribute: one with the value of the attribute initialised to its lower boundary while other attributes are still initialised to their means:

```
value[1+i] = tds->getAttribute(i)->getLowerBound();
tds->getTuple()->Fill(value);
```

and the other where the value of the attribute is initialised to its upper boundary while other attributes are still initialised to their means:

```
value[1+i] = tds->getAttribute(i)->getUpperBound();
tds->getTuple()->Fill(value);
```

In the first part of the function launchCodeFlowrateKeyOATMinMax, properties are set for each input variable. For example for the **rw** variable, the macro creates a TAttribute named **"rw"** bounded by 0.05 and 0.15:

```
tds->addAttribute( new TAttribute("rw", 0.05, 0.15));
```

A initial mean value value is then set for this TAttribute, so as to inform the PlanOATMinMax function.

```
tds->getAttribute("rw")->setMean(0.10);
```

Then, the TAttribute is linked to the first input file with:

```
tds->getAttribute("rw")->setFileKey(TString("flowrate_input_with_keys.in"), "Rw");
```

which will be understood by Uranie as the fact the **rw** variable has to be read in an input file with "key=value" format.

Then the design-of-experiments is generated by the PlanOATMinMax function.

```
PlanOATMinMax(tds);
```

The output file with values in rows _output_flowrate_withRow_.dat is instantiated as a TOutputFileRow.

```
TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
```

The variable **yhat** and **d** are then linked to this TOutputFileRow output file. But here, only **yhat** will be considered by Uranie as a variable of interest.

```
fout->addAttribute("yhat");
```

We set the code as being the **flowrate** execution with *"-k"* option:

```
TCode *mycode = new TCode(tds, "flowrate -k");
```

in which the "-k" option indicates that **flowrate** code has to find input files with "key=value" format.

Then the launcher is initialised with a TDataServer and the code is executed

```
TLauncher *tlch = new TLauncher(tds, mycode);
```

The launcher will execute the **flowrate -k** command for each of the seventeen patterns with the run method.

```
tlch->run();
```

XIV.4.17.3 Graph

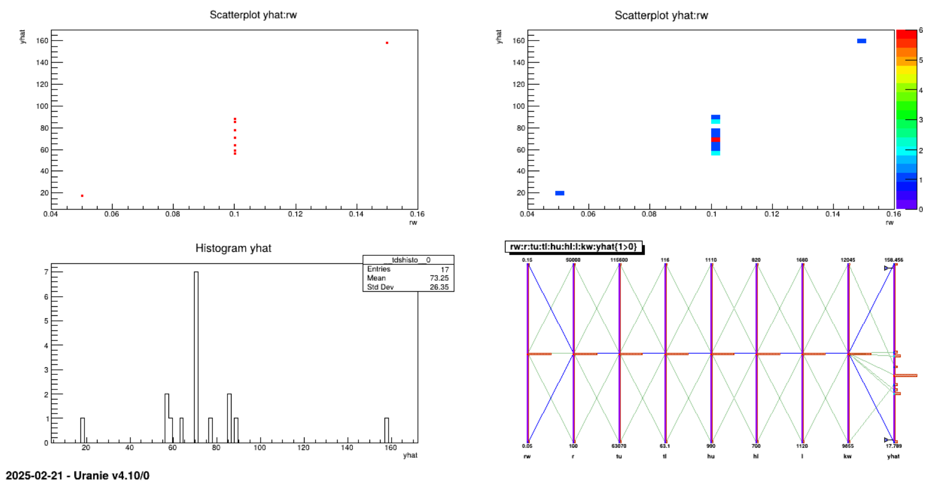


Figure XIV.33: Graph of the macro "launchCodeFlowrateKeyOATMinMax.C"

XIV.4.17.4 Console

The macro succeeded with no failure, using the following design-of-experiments;

```
Processing launchCodeFlowrateKeyOATMinMax.C...
******************************************************************************
*    Row   * tds * rw.r *   r.r *  tu.tu * tl.tl * hu.h * hl. *  l.l * kw.kw *
******************************************************************************
*        0 *   0 *  0.1 * 25050 *  89335 * 89.55 * 1050 * 760 * 1400 * 10950 *
*        1 *   1 * 0.05 * 25050 *  89335 * 89.55 * 1050 * 760 * 1400 * 10950 *
*        2 *   2 * 0.15 * 25050 *  89335 * 89.55 * 1050 * 760 * 1400 * 10950 *
*        3 *   3 *  0.1 *   100 *  89335 * 89.55 * 1050 * 760 * 1400 * 10950 *
*        4 *   4 *  0.1 * 50000 *  89335 * 89.55 * 1050 * 760 * 1400 * 10950 *
*        5 *   5 *  0.1 * 25050 *  63070 * 89.55 * 1050 * 760 * 1400 * 10950 *
*        6 *   6 *  0.1 * 25050 * 115600 * 89.55 * 1050 * 760 * 1400 * 10950 *
*        7 *   7 *  0.1 * 25050 *  89335 *  63.1 * 1050 * 760 * 1400 * 10950 *
*        8 *   8 *  0.1 * 25050 *  89335 *   116 * 1050 * 760 * 1400 * 10950 *
*        9 *   9 *  0.1 * 25050 *  89335 * 89.55 *  990 * 760 * 1400 * 10950 *
*       10 *  10 *  0.1 * 25050 *  89335 * 89.55 * 1110 * 760 * 1400 * 10950 *
*       11 *  11 *  0.1 * 25050 *  89335 * 89.55 * 1050 * 700 * 1400 * 10950 *
*       12 *  12 *  0.1 * 25050 *  89335 * 89.55 * 1050 * 820 * 1400 * 10950 *
*       13 *  13 *  0.1 * 25050 *  89335 * 89.55 * 1050 * 760 * 1120 * 10950 *
*       14 *  14 *  0.1 * 25050 *  89335 * 89.55 * 1050 * 760 * 1680 * 10950 *
*       15 *  15 *  0.1 * 25050 *  89335 * 89.55 * 1050 * 760 * 1400 *  9855 *
*       16 *  16 *  0.1 * 25050 *  89335 * 89.55 * 1050 * 760 * 1400 * 12045 *
******************************************************************************
```

XIV.4.18 Macro "launchCodeFlowrateFlagOATMinMax.C"

XIV.4.18.1 Objective

The objective of the macro is to evaluate the **flowrate** external code on a design-of-experiments generated by the function PlanOATMinMax where input file "flowrate_input_with_flags.in" contains "flags" and the output file "_output_flowrate_withRow_.dat" is with "values in rows" type. This code is described in section (Section IV.1.2.1). The function PlanOATMinMax consists in constructing a design around the nominal value for each attribute, except for one attribute whose value is set to one of its boundaries(minimum and then maximum). Doing this for the 8 input attributes and the central design, in which each attribute takes its nominal value leads to a seventeen patterns design.

The input file for this code is flowrate_input_with_flags.in with "flags" type. It looks like:

```
#
# INPUT FILE with FLAG for the "FLOWREATE" code
# \date   2008-04-22 12:55:17
#

new Implicit_Steady_State sch {
    frottement_paroi { @Rw@ @R@ }
    tinit               0.0
    tmax                1000000.
    nb_pas_dt_max       1500
    dt_min              @Hu@
    dt_max              @Hl@
    facsec              1000000.
    kW                  @Kw@
    information_Tu      Champ_Uniforme 1         @Tu@
    information_Tl      Champ_Uniforme 1         @Tl@
    information_L {
        precision  @L@
    }
    convergence {
        criterion relative_max_du_dt
        precision  @Rw@
    }

    stop_criterium {
        ch_abcsissa_hu 1050
        ch_ordinate_hl 770
        c_radius 1100
    }

    Solveur Newton3 {
        max_iter_matrice        1
        max_iter_implicite      1
        date                    5654321
        seuil_convg_implicite   1.e-6
        assemblage_implicite    10
        solveur_lineaire        BiCGS
            preconditionneur    ILU
            seuil_resol_implicite 1.e-5
    }
}
```

These files define the eight variables r_ω, r, T_u, T_l, H_u, H_l, L, K_ω needed to perform the execution of the command **flowrate -f**.

The output file, _output_flowrate_withRow_.dat when created, is of "values in rows" type. It looks like:

```
#COLUMN_NAMES: yhat | d

6.757218e+01  4.092561e+03
```

yhat and **d** have to be defined as output variables in the macro.

XIV.4.18.2 Macro Uranie

```
void PlanOATMinMax(TDataServer *tds)
{
  // Create the data tuple
  tds->createTuple();

  Int_t natt = tds->getNAttributes();
  // The number of attributes is incremented by one for the iterator
  const Int_t nn = natt+1;

  // Allocate the data vector to fill the tuple
  Double_t value[nn];

  // Init the iterator. It is always the first value.
  value[0] = 0;
  // Init the data vector with the nominal/mean values of attributes
  for(Int_t i=0;i<natt;i++)
    value[1+i] = tds->getAttribute(i)->getMean();

  // Fill the nominal/mean values : nominal pattern
  tds->getTuple()->Fill(value);

  // Loop on each attributes
  for(Int_t i=0;i<natt;i++) {
    // Case of min values
    value[1+i] = tds->getAttribute(i)->getLowerBound();
    value[0] += 1.0;
    // Fill the min value value for xi attribute and nominal values for other attributes
    tds->getTuple()->Fill(value);
    // Case of the max value
    value[1+i] = tds->getAttribute(i)->getUpperBound();
    value[0] += 1.0;
    // Fill the max value value for xi attribute and nominal values for other attributes
    tds->getTuple()->Fill(value);
    // Reset the nominal/mean value
    value[1+i] = tds->getAttribute(i)->getMean();
  }
}

void launchCodeFlowrateFlagOATMinMax()
{
  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsOATMinMaxFlowrateFlag", "OATMinMax Design of ↩
      experiments for Flowrate");

  // Add the eight attributes of the study with min, max and nominal values
  tds->addAttribute( new TAttribute("rw", 0.05, 0.15));
  tds->getAttribute("rw")->setMean(0.10);

  tds->addAttribute( new TAttribute("r", 100.0, 50000.0));
  tds->getAttribute("r")->setMean(25050.);

  tds->addAttribute( new TAttribute("tu", 63070.0, 115600.0));
  tds->getAttribute("tu")->setMean(89335.);

  tds->addAttribute( new TAttribute("tl", 63.1, 116.0));
  tds->getAttribute("tl")->setMean(89.55);

  tds->addAttribute( new TAttribute("hu", 990.0, 1110.0));
  tds->getAttribute("hu")->setMean(1050.);
  tds->addAttribute( new TAttribute("hl", 700.0, 820.0));
  tds->getAttribute("hl")->setMean(760.);

  tds->addAttribute( new TAttribute("l", 1120.0, 1680.0));
  tds->getAttribute("l")->setMean(1400.);

  tds->addAttribute( new TAttribute("kw", 9855.0, 12045.0));
  tds->getAttribute("kw")->setMean(10950.);

  // The input file
  TString sFileName = TString("flowrate_input_with_flags.in");

  // Set the input file with flag format and the flag for each input attributes
  tds->getAttribute("rw")->setFileFlag(sFileName, "@Rw@");
  tds->getAttribute("r")->setFileFlag(sFileName, "@R@");
  tds->getAttribute("tu")->setFileFlag(sFileName, "@Tu@");
  tds->getAttribute("tl")->setFileFlag(sFileName, "@Tl@");
  tds->getAttribute("hu")->setFileFlag(sFileName, "@Hu@");
  tds->getAttribute("hl")->setFileFlag(sFileName, "@Hl@");
  tds->getAttribute("l")->setFileFlag(sFileName, "@L@");
  tds->getAttribute("kw")->setFileFlag(sFileName, "@Kw@");

  // Build the Design of Experiments (DoE)
   PlanOATMinMax(tds);

  // Scan the data ( 17 = 1 + 2*8 patterns)
  tds->Scan("*","","colsize=5 col=3:4::6::4:3:4:");

  // Save the Design of experiments in an ASCII file
  tds->exportData("_flowrate_sampler_oat_.dat");

  // The output file of the code
  TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
  // The attribute in the output file
  fout->addAttribute(new TAttribute("yhat"));

  // Instanciation de mon code
  TCode *mycode = new TCode(tds, "flowrate -s -f");
  //mycode->setLog();
  // on ajoute le fichier de sortie du code
  mycode->addOutputFile( fout );

  // Lancement du code
  TLauncher *tlch = new TLauncher(tds, mycode);
  tlch->setSave();
  tlch->setClean();
  tlch->setDrawProgressBar(kFALSE);
  tlch->setVarDraw("yhat:rw","","");

  tlch->run();



  tds->exportData("_flowrate_sampler_launcher_oat_.dat");

  // Visualisation
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro launchCodeFlowrateFlagOATMinMax ↩
      ",5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  gStyle->SetPalette(1);
  pad->Divide(2, 2);
  pad->cd(1);
  tds->draw("yhat:rw");

  pad->cd(3); tds->draw("yhat");
  pad->cd(2); tds->draw("yhat:rw","","colz");

  pad->cd(4);
  // The parallel plot
  tds->draw("rw:r:tu:tl:hu:hl:l:kw:yhat","","para");
  TParallelCoord* para = (TParallelCoord*)gPad->GetListOfPrimitives()->FindObject(" ↩
      ParaCoord");

  // The output attribute
  TParallelCoordVar* axis = (TParallelCoordVar*)para->GetVarList()->FindObject("yhat");
  axis->AddRange(new TParallelCoordRange(axis,15.0,20.0));
  para->AddSelection("blue");
  para->GetCurrentSelection()->SetLineColor(kBlue);

  axis->AddRange(new TParallelCoordRange(axis,155.0,160.0));
  para->AddSelection("blue");
     para->GetCurrentSelection()->SetLineColor(kBlue);

}
```

PlanOATMinMax function builds the design-of-experiments from a TDataServer: for the first design and for each attribute of the TDataServer, values of attributes are initialised with their means.

```
for(Int_t i=0 ; i<natt ; i++)
value[1+i] = tds->getAttribute(i)->getMean();
```

These data are put in the TDataServer,

```
tds->getTuple()->Fill(value)
```

this is how the central point is constructed. Two designs are then built for every attribute: one with the value of the attribute initialised to its lower boundary while other attributes are still initialised to their means:

```
value[1+i] = tds->getAttribute(i)->getLowerBound();
tds->getTuple()->Fill(value);
```

and the other where the value of the attribute is initialised to its upper boundary while other attributes are still initialised to their means:

```
value[1+i] = tds->getAttribute(i)->getUpperBound();
tds->getTuple()->Fill(value);
```

In the first part of the function launchCodeFlowrateFlagOATMinMax, properties are set for each input variable. For example for the **Rw** variable, the macro creates a TAttribute named **"rw"** bounded by 0.05 and 0.15:

```
tds->addAttribute( new TAttribute("rw", 0.05, 0.15));
```

An initial mean value is then set for this TAttribute, so as to inform the PlanOATMinMax function.

```
tds->getAttribute("rw")->setMean(0.10);
```

Then, the TAttribute is linked to the input file with:

```
tds->getAttribute("rw")->setFileFlag(TString("flowrate_input_with_flags.in"), "@Rw@");
```

which is the same thing as writing:

```
tds->getAttribute("rw")->setFileKey(TString("flowrate_input_with_flags.in"), "@Rw@","", ↩
    TAttributeFileKey::kFlag);
```

which will be understood by Uranie as the fact the **rw** variable has to be read in an input file with "flags" format.

Then the design-of-experiments is generated by the `PlanOATMinMax` function.

```
PlanOATMinMax(tds);
```

We want the output file to be with values in rows. So the output file `_output_flowrate_withRow_.dat` is set as a `TOutputFileRow`.

```
TOutputFileRow *fout = new TOutputFileRow("_output_flowrate_withRow_.dat");
```

The variable **yhat** and **d** could then be linked to this `TOutputFileRow` output file. But here, only **yhat** will be considered.

```
fout->addAttribute("yhat");
```

We set the code as being the **flowrate** execution with *"-f"* option:

```
TCode *mycode = new TCode(tds, "flowrate -f");
```

in which the "-f" option indicates that **flowrate** code has to find input files with "flags" format.

The launcher will execute the **flowrate -f** command for each of the seventeen patterns. One sampling corresponds to one input file created by the launcher.

Then the launcher is initialised and launched with the `run` method

```
TLauncher *tlch = new TLauncher(tds, mycode);
tlch->run();
```

XIV.4.18.3 Graph

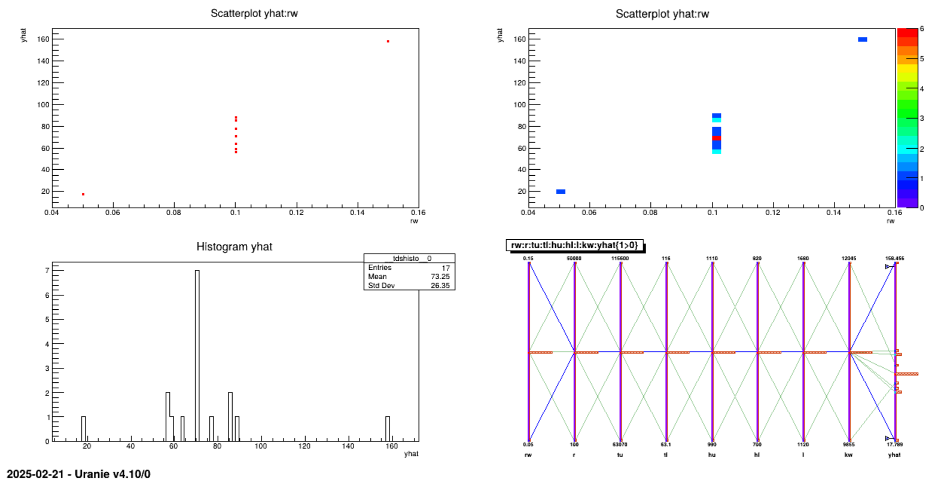


Figure XIV.34: Graph of the macro "`launchCodeFlowrateFlagOATMinMax.C`"

XIV.4.18.4 Console

The macro succeeded with no failure, using the following design-of-experiments:

```
Processing launchCodeFlowrateFlagOATMinMax.C...
******************************************************************************
*    Row   * tds * rw.r *   r.r *  tu.tu * tl.tl * hu.h * hl. *  l.l * kw.kw *
******************************************************************************
*        0 *   0 *  0.1 * 25050 *  89335 * 89.55 * 1050 * 760 * 1400 * 10950 *
*        1 *   1 * 0.05 * 25050 *  89335 * 89.55 * 1050 * 760 * 1400 * 10950 *
*        2 *   2 * 0.15 * 25050 *  89335 * 89.55 * 1050 * 760 * 1400 * 10950 *
*        3 *   3 *  0.1 *   100 *  89335 * 89.55 * 1050 * 760 * 1400 * 10950 *
*        4 *   4 *  0.1 * 50000 *  89335 * 89.55 * 1050 * 760 * 1400 * 10950 *
*        5 *   5 *  0.1 * 25050 *  63070 * 89.55 * 1050 * 760 * 1400 * 10950 *
*        6 *   6 *  0.1 * 25050 * 115600 * 89.55 * 1050 * 760 * 1400 * 10950 *
*        7 *   7 *  0.1 * 25050 *  89335 *  63.1 * 1050 * 760 * 1400 * 10950 *
*        8 *   8 *  0.1 * 25050 *  89335 *   116 * 1050 * 760 * 1400 * 10950 *
*        9 *   9 *  0.1 * 25050 *  89335 * 89.55 *  990 * 760 * 1400 * 10950 *
*       10 *  10 *  0.1 * 25050 *  89335 * 89.55 * 1110 * 760 * 1400 * 10950 *
*       11 *  11 *  0.1 * 25050 *  89335 * 89.55 * 1050 * 700 * 1400 * 10950 *
*       12 *  12 *  0.1 * 25050 *  89335 * 89.55 * 1050 * 820 * 1400 * 10950 *
*       13 *  13 *  0.1 * 25050 *  89335 * 89.55 * 1050 * 760 * 1120 * 10950 *
*       14 *  14 *  0.1 * 25050 *  89335 * 89.55 * 1050 * 760 * 1680 * 10950 *
*       15 *  15 *  0.1 * 25050 *  89335 * 89.55 * 1050 * 760 * 1400 *  9855 *
*       16 *  16 *  0.1 * 25050 *  89335 * 89.55 * 1050 * 760 * 1400 * 12045 *
******************************************************************************
```

XIV.4.19 Macro "launchCodeLevelEOutputColumn.C"

Warning
The **levele** command will be install on your machine only if a Fortran compiler has been found

XIV.4.19.1 Objective

The objective of this macro is to launch a code that will produce several numerical values for at least one attribute, when only once computation is performed. To do so, the **levele** use-case code will be used. It is a code that computes the dose emitted by radioactive sources, as a function of time, given a set of twelve input parameters. It provides three possible output formats, the one under consideration being here the *Column* one, corresponding to the `TOutputFileColumn` class. Here is an example of the output:

```
   20000.00       30000.00       40000.00       50000.00       60000.00       70000.00  ↩
           80000.00       90000.00       100000.0       200000.0       300000.0       ↩
       400000.0       500000.0       600000.0       700000.0       800000.0       900000.0  ↩
           1000000.       2000000.       3000000.       4000000.       5000000.       ↩
       6000000.       7000000.       8000000.       9000000.
  0.9223239E-37  0.4206201E-32  0.5129892E-29  0.9733449E-27  0.6020410E-25  0.1766700E-23  ↩
       0.3050058E-22  0.3543269E-21  0.3028644E-20  0.1192590E-14  0.6677703E-12  0.2881550E ↩
      -10  0.3190472E-09  0.1532743E-08  0.4212096E-08  0.7793285E-08  0.1077026E-07   ↩
      0.1192316E-07  0.1117317E-09 -0.9543977E-12  0.1073334E-19  0.2311560E-19  0.3043276E ↩
      -19  0.3529515E-19  0.4011028E-19  0.4548453E-19
   0.000000       0.000000       0.000000       0.000000       0.000000       0.000000  ↩
          0.1401298E-44  0.1261169E-42  0.9172900E-41  0.1422271E-29  0.4459172E-24   ↩
       0.8303329E-21  0.1017911E-18  0.2349300E-17  0.1774175E-16  0.6073529E-16  0.1159985E ↩
       -15  0.1421619E-15  0.1248398E-19  0.9108749E-24  0.1152050E-39  0.5343305E-39   ↩
       0.9261532E-39  0.1245747E-38  0.1608835E-38  0.2068842E-38
```

XIV.4.19.2 Macro Uranie

```
{
  // OS abstraction
  string which_levele =
    string(gSystem->GetBuildArch()) == "win64" ? "where levele" : "which levele";

  //Exit if levele not found
  if(gSystem->Exec(which_levele.c_str()))
    exit(-1);

  //Create DataServer and add input attributes
  TDataServer *tds1 = new TDataServer("tds1", "levelE usecase");
  tds1->addAttribute( new TUniformDistribution("t",100,1000) );
  tds1->addAttribute( new TLogUniformDistribution("kl",0.001,0.01) );
  tds1->addAttribute( new TLogUniformDistribution("kc",0.000001,0.00001) );
  tds1->addAttribute( new TLogUniformDistribution("v1",0.001,0.1) );
  tds1->addAttribute( new TUniformDistribution("l1",100,500) );
  tds1->addAttribute( new TUniformDistribution("r1",1,5) );
  tds1->addAttribute( new TUniformDistribution("rc1",3,30) );
  tds1->addAttribute( new TLogUniformDistribution("v2",0.01,0.1) );
  tds1->addAttribute( new TUniformDistribution("l2",50,200) );
  tds1->addAttribute( new TUniformDistribution("r2",1,5) );
  tds1->addAttribute( new TUniformDistribution("rc2",3,30) );
  tds1->addAttribute( new TLogUniformDistribution("w",100000,10000000) );

  //Tell the code where to find attribute value in input file
  TString sJDD = "levelE_input_with_values_rows.in";
  tds1->getAttribute("t")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("kl")->setFileKey(sJDD,"", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("kc")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("v1")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("l1")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("r1")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("rc1")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("v2")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("l2")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("r2")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("rc2")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("w")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);

  // Create DOE
  Int_t ns = 100;
  TSampling *samp = new TSampling(tds1, "lhs", ns);
  samp->generateSample();

  //How to read ouput files
  TOutputFileColumn *_fout = new URANIE::Launcher::TOutputFileColumn(" ↩
     _output_levelE_withColumn_.dat");
  _fout->addAttribute(new TAttribute("tps",TAttribute::kVector));
  _fout->addAttribute(new TAttribute("y",TAttribute::kVector));
  _fout->addAttribute(new TAttribute("z",TAttribute::kVector));

  //Creation of TCode
  TCode *tc1 = new URANIE::Launcher::TCode(tds1, "levele 2> /dev/null");
  tc1->addOutputFile( _fout );

  //Run the code
  TLauncher *tl1 = new URANIE::Launcher::TLauncher(tds1, tc1);
  tl1->run();

  //Draw the results
  TCanvas *Can = new TCanvas("Can","Can",1);
  Can->SetGrid(); Can->SetLogx();

  double mean[26], stand[26];
  tds1->computeStatistic("y");
  for(unsigned int i=0; i<26; i++)
    {
      mean[i] = tds1->getAttribute("y")->getMean(i);
      stand[i] = tds1->getAttribute("y")->getStd(i);
    }

  double tps[26]={20000,30000,40000,50000,60000,70000,80000,90000,100000,  ↩
      200000,300000,400000,500000,600000,700000,800000,900000, 1e+06,2e+06,3e+06,4e+06,5e ↩
      +06,6e+06,7e+06,8e+06,9e+06};
  TGraphErrors *gr = new TGraphErrors(26,tps,mean,0,stand);
  gr->Draw("A*");
  gr->SetTitle("");
  gr->GetXaxis()->SetTitle("Time"); gr->GetYaxis()->SetTitle("<y>");

}
```

The **levele** external code is located in the bin directory.

When looking at the code and comparing it to an usual launcher job, the organisation is completely transparent. The only noticeable (and compulsory) thing to do is to change the default type of the attribute read at the end of the job. This is done in this line:

```
_fout->addAttribute(new URANIE::DataServer::TAttribute("y", TAttribute::kVector));
```

where the output attribute is provided, changing its nature to a vector, thanks to the second argument of the `TAttribute` constructor from the default (`kReal`) to the desired nature (`kVector`). Once this is done, this information is broadcast internally to the code that knows how to deal with this type of attribute.

XIV.4.19.3 Graph

The results of the previous macro is shown in Figure XIV.35, that displays the average of the hundred dose curve, over the time.

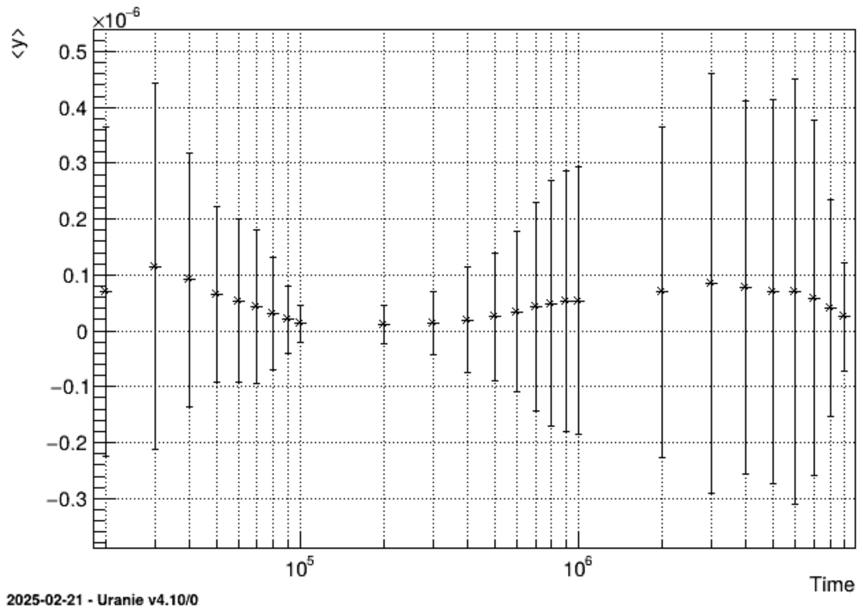


Figure XIV.35: Graph of the macro "`launchCodeLevelEOutputColumn.C`"

XIV.4.20 Macro "`launchCodeLevelEOutputRow.C`"

Warning
The **levele** command will be installed on your machine only if a Fortran compiler is found

XIV.4.20.1 Objective

The objective of this macro is to launch a code that will produce several numerical values for at least one attribute, when only once computation is performed. To do so, the **levele** use-case code will be used. It is a code that compute the dose emitted by radioactive sources, as a function of time, given a set of twelve inputs parameters. It provides three possible output formats, the one under consideration being here the *Row* one, corresponding to the `TOutputFileRow` class. Here is an example of the output:

```
#COLUMN_NAMES: tps | y | z

   20000.00      0.9223239E-37   0.000000
   30000.00      0.4206201E-32   0.000000
   40000.00      0.5129892E-29   0.000000
   50000.00      0.9733449E-27   0.000000
   60000.00      0.6020410E-25   0.000000
   70000.00      0.1766700E-23   0.000000
   80000.00      0.3050058E-22  0.1401298E-44
   90000.00      0.3543269E-21  0.1261169E-42
   100000.0      0.3028644E-20  0.9172900E-41
   200000.0      0.1192590E-14  0.1422271E-29
   300000.0      0.6677703E-12  0.4459172E-24
   400000.0      0.2881550E-10  0.8303329E-21
   500000.0      0.3190472E-09  0.1017911E-18
   600000.0       0.1532743E-08  0.2349300E-17
   700000.0       0.4212096E-08  0.1774175E-16
   800000.0       0.7793285E-08  0.6073529E-16
   900000.0       0.1077026E-07  0.1159985E-15
   1000000.       0.1192316E-07  0.1421619E-15
   2000000.       0.1117317E-09  0.1248398E-19
   3000000.      -0.9543977E-12  0.9108749E-24
   4000000.       0.1073334E-19  0.1152050E-39
   5000000.       0.2311560E-19  0.5343305E-39
   6000000.       0.3043276E-19  0.9261532E-39
   7000000.       0.3529515E-19  0.1245747E-38
   8000000.       0.4011028E-19  0.1608835E-38
   9000000.       0.4548453E-19  0.2068842E-38
```

XIV.4.20.2 Macro Uranie

```
{
  // OS abstraction
  string which_levele =
    string(gSystem->GetBuildArch()) == "win64" ? "where levele" : "which levele";

  //Exit if levele not found
  if(gSystem->Exec(which_levele.c_str()))
    exit(-1);

  //Create DataServer and add input attributes
  TDataServer *tds1 = new TDataServer("tds1", "levelE usecase");
  tds1->addAttribute( new TUniformDistribution("t",100,1000) );
  tds1->addAttribute( new TLogUniformDistribution("kl",0.001,0.01) );
  tds1->addAttribute( new TLogUniformDistribution("kc",0.000001,0.00001) );
  tds1->addAttribute( new TLogUniformDistribution("v1",0.001,0.1) );
  tds1->addAttribute( new TUniformDistribution("l1",100,500) );
  tds1->addAttribute( new TUniformDistribution("r1",1,5) );
  tds1->addAttribute( new TUniformDistribution("rc1",3,30) );
  tds1->addAttribute( new TLogUniformDistribution("v2",0.01,0.1) );
  tds1->addAttribute( new TUniformDistribution("l2",50,200) );
  tds1->addAttribute( new TUniformDistribution("r2",1,5) );
  tds1->addAttribute( new TUniformDistribution("rc2",3,30) );
  tds1->addAttribute( new TLogUniformDistribution("w",100000,10000000) );

  //Tell the code where to find attribute value in input file
  TString sJDD = "levelE_input_with_values_rows.in";
  tds1->getAttribute("t")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("kl")->setFileKey(sJDD,"", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("kc")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("v1")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("l1")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("r1")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("rc1")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("v2")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("l2")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("r2")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("rc2")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("w")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);

  // Create DOE
  Int_t ns = 100;
  TSampling *samp = new TSampling(tds1, "lhs", ns);
  samp->generateSample();
  //How to read ouput files
  TOutputFileRow *_fout = new URANIE::Launcher::TOutputFileRow("_output_levelE_withRow_.dat ↩
      ");
  _fout->addAttribute(new TAttribute("tps",TAttribute::kVector));
  _fout->addAttribute(new TAttribute("y",TAttribute::kVector));
  _fout->addAttribute(new TAttribute("z",TAttribute::kVector));

  //Creation of TCode
  TCode *tc1 = new URANIE::Launcher::TCode(tds1, "levele 2> /dev/null");
  tc1->addOutputFile( _fout );

  //Run the code
  TLauncher *tl1 = new URANIE::Launcher::TLauncher(tds1, tc1);
  tl1->run();

  //Draw the results
  TCanvas *Can = new TCanvas("Can","Can",1);
  Can->SetGrid(); Can->SetLogx();

  double mean[26], stand[26];
  tds1->computeStatistic("y");
  for(unsigned int i=0; i<26; i++)
    {
      mean[i] = tds1->getAttribute("y")->getMean(i);
      stand[i] = tds1->getAttribute("y")->getStd(i);
    }

  double tps[26]={20000,30000,40000,50000,60000,70000,80000,90000,100000,  ↩
      200000,300000,400000,500000,600000,700000,800000,900000, 1e+06,2e+06,3e+06,4e+06,5e ↩
      +06,6e+06,7e+06,8e+06,9e+06};
  TGraphErrors *gr = new TGraphErrors(26,tps,mean,0,stand);
  gr->Draw("A*");
  gr->SetTitle("");
  gr->GetXaxis()->SetTitle("Time"); gr->GetYaxis()->SetTitle("<y>");

}
```

The **levele** external code is located in the bin directory.

When looking at the code and comparing it to an usual launcher job, the organisation is completely transparent. The only noticeable (and compulsory) thing to do is to change the default type of the attribute read at the end of the job. This is done in this line:

```
_fout->addAttribute(new URANIE::DataServer::TAttribute("y", TAttribute::kVector));
```

where the output attribute is provided, changing its nature to a vector, thanks to the second argument of the `TAttribute` constructor from the default (`kReal`) to the desired nature (`kVector`). Once this is done, this information is broadcast internally to the code that knows how to deal with this type of attribute.

XIV.4.20.3 Graph

The results of the previous macro is shown in Figure XIV.36, that displays the average of the hundred dose curve, over the time.

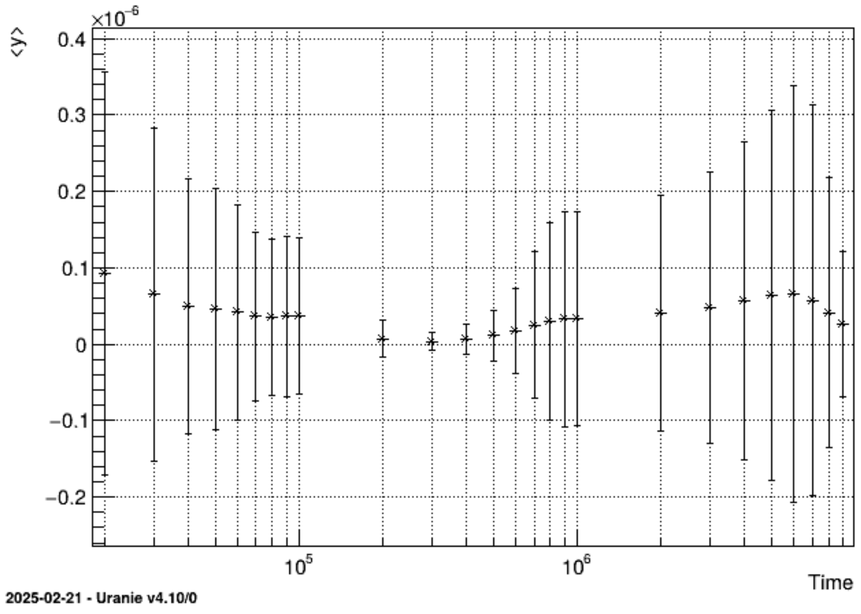


Figure XIV.36: Graph of the macro "launchCodeLevelEOutputRow.C"

XIV.4.21 Macro "launchCodeLevelEOutputKey.C"

Warning
The **levele** command will be installed on your machine only if a Fortran compiler is found

XIV.4.21.1 Objective

The objective of this macro is to launch a code that will produce several numerical values for at least one attribute, when only once computation is performed. To do so, the **levele** use-case code will be used. It is a code that compute the dose emitted by radioactive sources, as a function of time, given a set of twelve inputs parameters. It provides three possible output formats, the one under consideration being here the *Key* one, corresponding to the `TOutputFileKey` class. Here is an example of the output:

```
time =    20000.00
y =   0.9223239E-37
z =    0.000000
time =    30000.00
y =   0.4206201E-32
z =    0.000000
time =    40000.00
y =   0.5129892E-29
z =    0.000000
time =    50000.00
y =   0.9733449E-27
z =    0.000000
time =    60000.00
y =   0.6020410E-25
z =    0.000000
```

```
time =    70000.00
y =   0.1766700E-23
z =    0.000000
time =    80000.00
y =   0.3050058E-22
z =   0.1401298E-44
time =    90000.00
y =   0.3543269E-21
z =   0.1261169E-42
time =    100000.0
y =   0.3028644E-20
z =   0.9172900E-41
time =    200000.0
y =   0.1192590E-14
z =   0.1422271E-29
time =    300000.0
y =   0.6677703E-12
z =   0.4459172E-24
time =    400000.0
y =   0.2881550E-10
z =   0.8303329E-21
time =    500000.0
y =   0.3190472E-09
z =   0.1017911E-18
time =    600000.0
y =   0.1532743E-08
z =   0.2349300E-17
time =    700000.0
y =   0.4212096E-08
z =   0.1774175E-16
time =    800000.0
y =   0.7793285E-08
z =   0.6073529E-16
time =    900000.0
y =   0.1077026E-07
z =   0.1159985E-15
time =    1000000.
y =   0.1192316E-07
z =   0.1421619E-15
time =    2000000.
y =   0.1117317E-09
z =   0.1248398E-19
time =    3000000.
y =  -0.9543977E-12
z =   0.9108749E-24
time =    4000000.
y =   0.1073334E-19
z =   0.1152050E-39
time =    5000000.
y =   0.2311560E-19
z =   0.5343305E-39
time =    6000000.
y =   0.3043276E-19
z =   0.9261532E-39
time =    7000000.
y =   0.3529515E-19
z =   0.1245747E-38
time =    8000000.
y =   0.4011028E-19
z =   0.1608835E-38
time =    9000000.
y =   0.4548453E-19
```

```
z =   0.2068842E-38
```

XIV.4.21.2 Macro Uranie

```
{
  // OS abstraction
  string which_levele =
    string(gSystem->GetBuildArch()) == "win64" ? "where levele" : "which levele";

  //Exit if levele not found
  if(gSystem->Exec(which_levele.c_str()))
    exit(-1);

  //Create DataServer and add input attributes
  TDataServer *tds1 = new TDataServer("tds1", "levelE usecase");
  tds1->addAttribute( new TUniformDistribution("t",100,1000) );
  tds1->addAttribute( new TLogUniformDistribution("kl",0.001,0.01) );
  tds1->addAttribute( new TLogUniformDistribution("kc",0.000001,0.00001) );
  tds1->addAttribute( new TLogUniformDistribution("v1",0.001,0.1) );
  tds1->addAttribute( new TUniformDistribution("l1",100,500) );
  tds1->addAttribute( new TUniformDistribution("r1",1,5) );
  tds1->addAttribute( new TUniformDistribution("rc1",3,30) );
  tds1->addAttribute( new TLogUniformDistribution("v2",0.01,0.1) );
  tds1->addAttribute( new TUniformDistribution("l2",50,200) );
  tds1->addAttribute( new TUniformDistribution("r2",1,5) );
  tds1->addAttribute( new TUniformDistribution("rc2",3,30) );
  tds1->addAttribute( new TLogUniformDistribution("w",100000,10000000) );

  //Tell the code where to find attribute value in input file
  TString sJDD = "levelE_input_with_values_rows.in";
  tds1->getAttribute("t")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("kl")->setFileKey(sJDD,"", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("kc")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("v1")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("l1")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("r1")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("rc1")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("v2")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("l2")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("r2")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("rc2")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);
  tds1->getAttribute("w")->setFileKey(sJDD, "", "%e",TAttributeFileKey::kNewRow);

  // Create DOE
  Int_t ns = 100;
  TSampling *samp = new TSampling(tds1, "lhs", ns);
  samp->generateSample();

  //How to read ouput files
  TOutputFileKey *_fout = new URANIE::Launcher::TOutputFileKey("_output_levelE_withKey_.dat ↩
      ");
  _fout->addAttribute(new TAttribute("tps",TAttribute::kVector));
  _fout->addAttribute(new TAttribute("y",TAttribute::kVector));
  _fout->addAttribute(new TAttribute("z",TAttribute::kVector));

  //Creation of TCode
  TCode *tc1 = new URANIE::Launcher::TCode(tds1, "levele 2> /dev/null");
  tc1->addOutputFile( _fout );

  //Run the code
  TLauncher *tl1 = new URANIE::Launcher::TLauncher(tds1, tc1);
  tl1->run();

  //Draw the results
  TCanvas *Can = new TCanvas("Can","Can",1);
  Can->SetGrid(); Can->SetLogx();

  double mean[26], stand[26];
  tds1->computeStatistic("y");
  for(unsigned int i=0; i<26; i++)
    {
      mean[i] = tds1->getAttribute("y")->getMean(i);
      stand[i] = tds1->getAttribute("y")->getStd(i);
    }

  double tps[26]={20000,30000,40000,50000,60000,70000,80000,90000,100000,  ↩
      200000,300000,400000,500000,600000,700000,800000,900000, 1e+06,2e+06,3e+06,4e+06,5e ↩
      +06,6e+06,7e+06,8e+06,9e+06};
  TGraphErrors *gr = new TGraphErrors(26,tps,mean,0,stand);
  gr->Draw("A*");
  gr->SetTitle("");
  gr->GetXaxis()->SetTitle("Time"); gr->GetYaxis()->SetTitle("<y>");

}
```

The **levele** external code is located in the bin directory.

When looking at the code and comparing it to an usual launcher job, the organisation is completely transparent. The only noticeable (and compulsory) thing to do is to change the default type of the attribute read at the end of the job. This is done in this line:

```
_fout->addAttribute(new URANIE::DataServer::TAttribute("y", TAttribute::kVector));
```

where the output attribute is provided, changing its nature to a vector, thanks to the second argument of the `TAttribute` constructor from the default (`kReal`) to the desired nature (`kVector`). Once this is done, this information is broadcast internally to the code that knows how to deal with this type of attribute.

XIV.4.21.3 Graph

The results of the previous macro is shown in Figure XIV.37, that displays the average of the hundred dose curve, over the time.

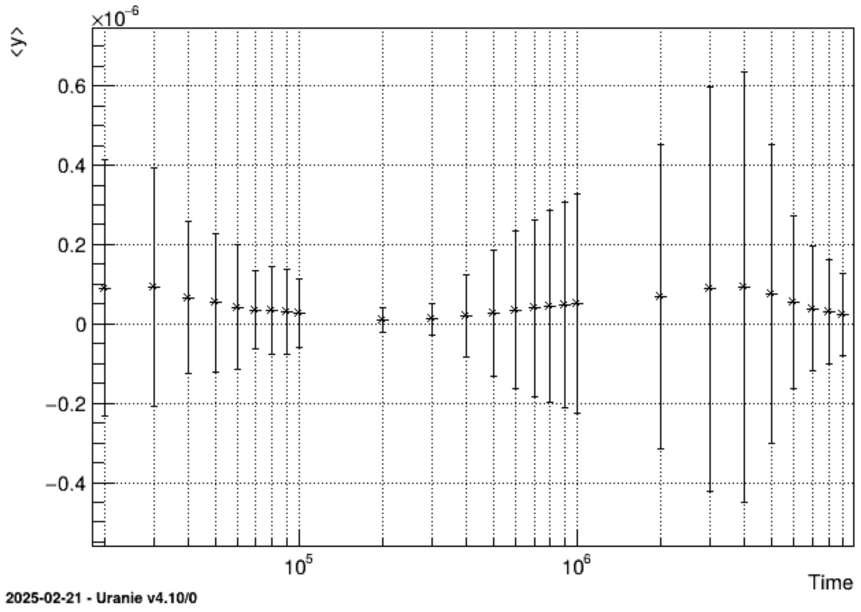


Figure XIV.37: Graph of the macro "`launchCodeLevelEOutputKey.C`"

XIV.4.22 Input/Output with vector and string: introduction to macros with multitype

In order to test the possibility of reading vectors and strings (for jobs that produce those kind of results) but also to re-write them in files when they are requested by a code (as an input) the **multitype** code has been created. It is just a dummy piece of code that generates outputs containing vectors and strings along with double-precision results, and re-read them to use the information contained and produce a new flat result.

XIV.4.22.1 Producing outputs

Special flags have been added to the **multitype** code aiming to produce an output containing:

- 2 words: w1 and w2, at first and last position, drawn randomly from a list that goes from "zero" to "nine".
- 2 vectors: v1 and v2, at second and third position, filled with N_{v_1} and N_{v_2}, doubles that follow two different gaussian drawings. These number N_{v_1} and N_{v_2} are drawn randomly between 1 and 15 and might then changed from one event to another.
- 1 float, f1, at fourth position, drawn following a gaussian.

The code can produce different kind of output files depending on the way it is called:

- `multitype -mtKey`: produces 2 output files (`_output_multitype_mt_Key_.dat` and `_output_multitype_mt_Key_condensate_.dat`) written in Key format
- `multitype -mtRow`: produces an output file (`_output_multitype_mt_Row_.dat`) written in Row format
- `multitype -mtCol`: produces an output file (`_output_multitype_mt_Column_.dat`) written in Column format

- `multitype -mtDS`: produces an output file (`_output_multitype_mt_DataServer_.dat`) written in DataServer format
- `multitype -mtXML`: produces an output file (`_output_multitype_mt_.xml`) written in XML format

The code uses an input file called "`multitype_input.dat`" containing a single double that corresponds to the seed used to generate the outputs. This is the only attribute needed when this *-mt* option is used for **multitype**.

XIV.4.22.2 Reading inputs

Another special flag has been added to the **multitype** code aiming to analyse the output of the step described previously. The idea is to check that vectors and strings are read and replaced carefully as input for the new operation which produces two double-precision outputs: thev1 and thev2. Their values is set to 987654321 by default and when the code is run the idea is:

- to recover the word w1 and the vector v1.
- to get the number that corresponds to the words w1 ("two" == 2).
- to associate thev1 to the w1-Th value of the vector v1 if it exists (and set it to -123456789 otherwise).

This procedure being repeated for w2 and v2 as well.

The code can produce different kinds of output files, from different input files, depending on the way it is called:

- `multitype -ReadmtKey`: produces an output files (`_output_multitype_readmt_Key_.dat`) by reading the input variable in `_output_multitype_mt_Key_condensate_.dat`).
- `multitype -ReadmtRow`: produces an output file (`_output_multitype_readmt_Row_.dat`) by reading the input variable in `_output_multitype_mt_Row_.dat`).
- `multitype -ReadmtCol`: produces an output file (`_output_multitype_readmt_Column_.dat`) by reading the input variable in `_output_multitype_mt_Column_.dat`).
- `multitype -ReadmtDS`: produces an output file (`_output_multitype_readmt_DataServer_.dat`) by reading the input variable in `_output_multitype_mt_DataServer_.dat`).
- `multitype -ReadmtXML`: produces an output file (`_output_multitype_readmt_.xml`) by reading the input variable in `_output_multitype_mt_.xml`).

XIV.4.23 Macro "launchCodeMultiTypeKey.C"

XIV.4.23.1 Objective

The objective of this macro is to test the case where vectors and strings are produced as outputs, using the code described in Section XIV.4.22.1, with a Key format, obtained by doing:

```
multitype -mtKey
```

The resulting output file, named `_output_multitype_mt_Key_.dat` looks like:

```
w1 = nine
v1 = -0.512095
v1 = 0.039669
v1 = -1.3834
v1 = 1.37667
v1 = 0.220672
v1 = 0.633267
v1 = 1.37027
v1 = -0.765636
v2 = 14.1981
v2 = 14.0855
v2 = 10.7848
v2 = 9.45476
v2 = 9.17308
v2 = 6.60804
v2 = 10.0711
v2 = 14.1761
v2 = 10.318
v2 = 12.5095
v2 = 15.6614
v2 = 10.3452
v2 = 9.41101
v2 = 7.47887
f1 = 32.2723
w2 = eight
```

XIV.4.23.2 Macro Uranie

```
{
  //Create dataserver with the seed attribute
  TDataServer *tds = new TDataServer("foo","pouet");
  tds->addAttribute( new TUniformDistribution("seed",0,100000));
  tds->getAttribute("seed")->setFileKey("multitype_input.dat","", "%e",TAttributeFileKey:: ↩
      kNewRow);

  //Create DOE
  TSampling *tsam = new TSampling(tds,"lhs",100);
  tsam->generateSample();

  //Precise output and create Code
  TOutputFileKey *out = new TOutputFileKey("_output_multitype_mt_Key_.dat");
  out->addAttribute(new TAttribute("w1",TAttribute::kString));
  out->addAttribute(new TAttribute("w2",TAttribute::kString));
  out->addAttribute(new TAttribute("v1",TAttribute::kVector));
  out->addAttribute(new TAttribute("v2",TAttribute::kVector));
  out->addAttribute(new TAttribute("f1"));
  TCode *myc = new URANIE::Launcher::TCode(tds, "multitype -mtKey");
  myc->addOutputFile( out );

  //Create TLauncher and run it
  TLauncher *tlau = new URANIE::Launcher::TLauncher(tds, myc);
  tlau->run();

  //Produce control plot
  TCanvas *Can = new TCanvas("Can","Can",10,10,1000,800);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw(); pad->cd();
  tds->drawPairs("w1:v1:v2:f1:w2");

}
```

When looking at the code and comparing it to an usual launcher job, the organisation is completely transparent. A dataserver is created with a simple input (the seed) drawn as an uniform distribution. The design-of-experiments is generated and the `TOutputFileKey` is created and filled with new attributes. This is the important part, where nature of the attributes are precised:

```
TOutputFileKey *out = new TOutputFileKey("_output_multitype_mt_Key_.dat");
out->addAttribute(new URANIE::DataServer::TAttribute("w1",TAttribute::kString));
out->addAttribute(new URANIE::DataServer::TAttribute("w2",TAttribute::kString));
out->addAttribute(new URANIE::DataServer::TAttribute("v1",TAttribute::kVector));
out->addAttribute(new URANIE::DataServer::TAttribute("v2",TAttribute::kVector));
out->addAttribute(new URANIE::DataServer::TAttribute("f1"));
```

The rest is very common and a pair plot is performed to check that the reading of the output went well (see following plot).

XIV.4.23.3 Graph

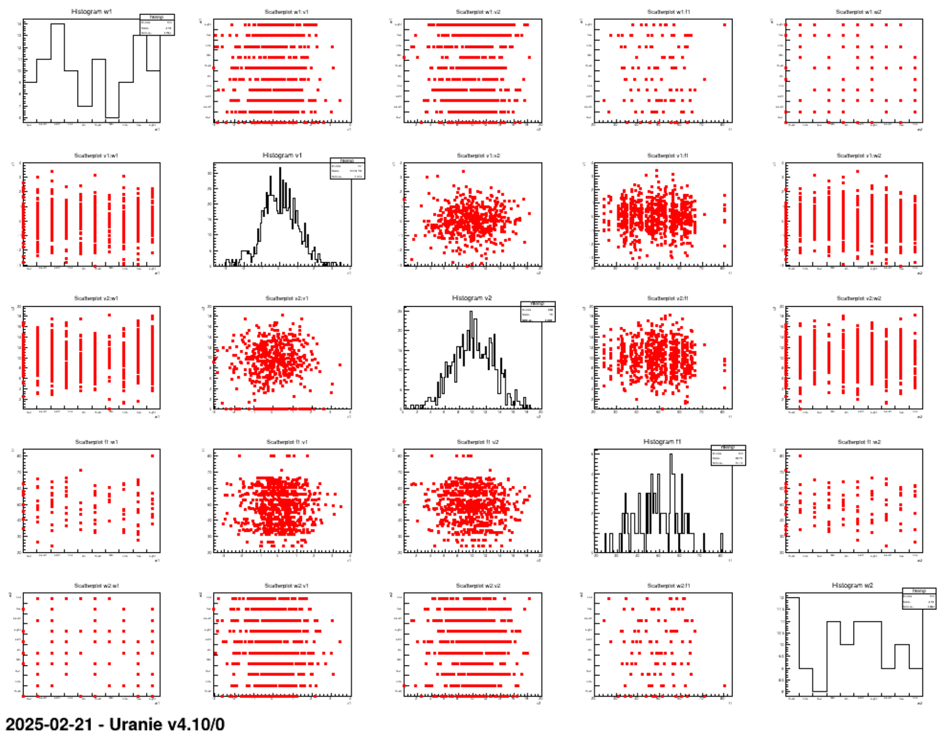


Figure XIV.38: Graph of the macro "`launchCodeMultiTypeKey.C`"

XIV.4.24 Macro "`launchCodeMultiTypeKeyCondensate.C`"

XIV.4.24.1 Objective

The objective of this macro is to test the case where vectors and strings are produced as outputs, using the code described in Section XIV.4.22.1, with a Key format, obtained by doing:

```
multitype -mtKey
```

The resulting output file, named `_output_multitype_mt_Key_condensate_.dat` looks like:

```
w1 = nine
v1 = [ -0.512095,0.039669,-1.3834,1.37667,0.220672,0.633267,1.37027,-0.765636 ]
v2 = [  ↩
    14.1981,14.0855,10.7848,9.45476,9.17308,6.60804,10.0711,14.1761,10.318,12.5095,15.6614,10.3452,9.4110
     ]
f1 = 32.2723
w2 = eight
```

It is a much more readable format than the previously defined one. Note that the vector-values are comma-separated without blanks. This has been discussed in Section IV.3.1.1 and is compulsory because of the way the files (in the Launcher module) are read.

XIV.4.24.2 Macro Uranie

```
{
  //Create dataserver with the seed attribute
  TDataServer *tds = new TDataServer("foo","pouet");
  tds->addAttribute( new TUniformDistribution("seed",0,100000));
  tds->getAttribute("seed")->setFileKey("multitype_input.dat","", "%e",TAttributeFileKey:: ↩
      kNewRow);

  //Create DOE
  TSampling *tsam = new TSampling(tds,"lhs",100);
  tsam->generateSample();

  //Precise output and create Code
  TOutputFileKey *out = new TOutputFileKey("_output_multitype_mt_Key_condensate_.dat");
  out->addAttribute(new TAttribute("w1",TAttribute::kString));
  out->addAttribute(new TAttribute("w2",TAttribute::kString));
  out->addAttribute(new TAttribute("v1",TAttribute::kVector));
  out->addAttribute(new TAttribute("v2",TAttribute::kVector));
  out->addAttribute(new TAttribute("f1"));
  //Specify that the output will be separated by commas and that [ and ] have to be  ↩
      considered as separator (== ignored)
  out->setVectorProperties("[",",","]");

  TCode *myc = new URANIE::Launcher::TCode(tds, "multitype -mtKey");
  myc->addOutputFile( out );

  //Create TLauncher and run it
  TLauncher *tlau = new URANIE::Launcher::TLauncher(tds, myc);
  tlau->run();

  //Produce control plot
  TCanvas *Can = new TCanvas("Can","Can",10,10,1000,800);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw(); pad->cd();
  tds->drawPairs("w1:v1:v2:f1:w2");

}
```

When looking at the code and comparing it to an usual launcher job, the organisation is completely transparent. Actually it is the exact same code as in Section XIV.4.24, up to this line:

```
//Specify that the output will be separated by commas and that [ and ] have to be  ↩
    considered as separator (== ignored)
out->setVectorProperties("[",",","]");
```

Thanks to this line, the output file parser will know that all vector values have to be read at once, and it will know how to separate them. This is the line to modify in order to go from key format describe in Section XIV.4.24 to a condensate one. The rest is very common and a pair plot is performed to check that the reading of the output went well (see following plot).

XIV.4.24.3 Graph

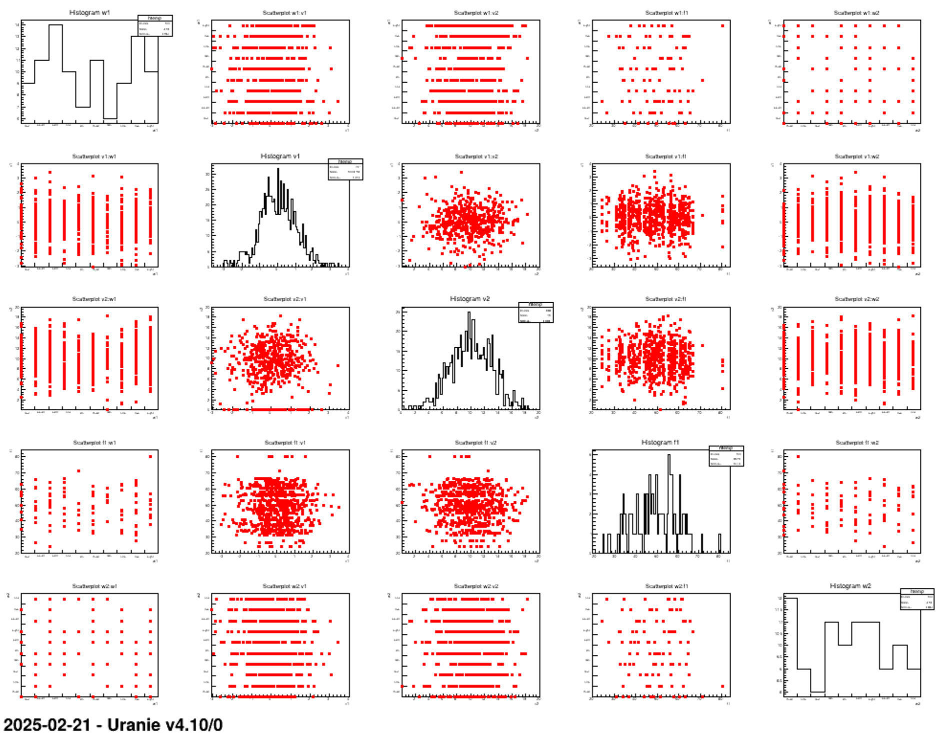


Figure XIV.39: Graph of the macro "`launchCodeMultiTypeKeyCondensate.C`"

XIV.4.25 Macro "`launchCodeMultiTypeDataServer.C`"

XIV.4.25.1 Objective

The objective of this macro is to test the case where vectors and strings are produced as outputs, using the code described in Section XIV.4.22.1, with a DataServer format, obtained by doing:

```
multitype -mtDS
```

The resulting output file, named `_output_multitype_mt_DataServer_.dat` looks like:

```
#COLUMN_NAMES: w1|v1|v2|f1|w2
#COLUMN_TYPES: S|V|V|D|S

nine -0.512095,0.039669,-1.3834,1.37667,0.220672,0.633267,1.37027,-0.765636  ↩
    14.1981,14.0855,10.7848,9.45476,9.17308,6.60804,10.0711,14.1761,10.318,12.5095,15.6614,10.3452,9
     32.2723 eight
```

XIV.4.25.2 Macro Uranie

```
{
  //Create dataserver with the seed attribute
  TDataServer *tds = new TDataServer("foo","pouet");
  tds->addAttribute( new TUniformDistribution("seed",0,100000));
  tds->getAttribute("seed")->setFileKey("multitype_input.dat","", "%e",TAttributeFileKey:: ↩
      kNewRow);

  //Create DOE
  TSampling *tsam = new TSampling(tds,"lhs",100);
  tsam->generateSample();

  //Precise output and create Code
  TOutputFileDataServer *out = new TOutputFileDataServer("_output_multitype_mt_DataServer_. ↩
      dat");
  out->addAttribute(new TAttribute("w1",TAttribute::kString));
  out->addAttribute(new TAttribute("w2",TAttribute::kString));
  out->addAttribute(new TAttribute("v1",TAttribute::kVector));
  out->addAttribute(new TAttribute("v2",TAttribute::kVector));
  out->addAttribute(new TAttribute("f1"));
  TCode *myc = new URANIE::Launcher::TCode(tds, "multitype -mtDS");
  myc->addOutputFile( out );

  //Create TLauncher and run it
  TLauncher *tlau = new URANIE::Launcher::TLauncher(tds, myc);
  tlau->run();

  //Produce control plot
  TCanvas *Can = new TCanvas("Can","Can",10,10,1000,800);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw(); pad->cd();
  tds->drawPairs("w1:v1:v2:f1:w2");

}
```

When looking at the code and comparing it to an usual launcher job, the organisation is completely transparent. A dataserver is created with a simple input (the seed) drawn as an uniform distribution. The design-of-experiments is generated and the `TOutputFileDataServer` is created and filled with new attributes. This is the important part, where nature of the attributes are precised:

```
TOutputFileDataServer *out = new TOutputFileDataServer("_output_multitype_mt_DataServer_. ↩
    dat");
out->addAttribute(new URANIE::DataServer::TAttribute("w1",TAttribute::kString));
out->addAttribute(new URANIE::DataServer::TAttribute("w2",TAttribute::kString));
out->addAttribute(new URANIE::DataServer::TAttribute("v1",TAttribute::kVector));
out->addAttribute(new URANIE::DataServer::TAttribute("v2",TAttribute::kVector));
out->addAttribute(new URANIE::DataServer::TAttribute("f1"));
```

The rest is very common and a pair plot is performed to check that the reading of the output went well (see following plot).

XIV.4.25.3 Graph

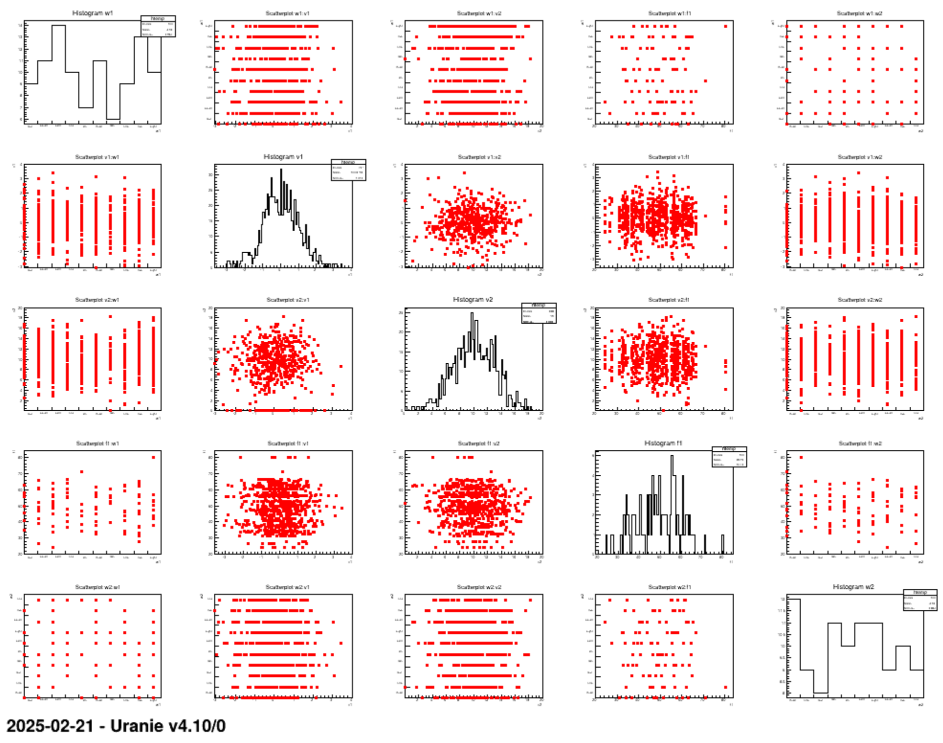


Figure XIV.40: Graph of the macro "launchCodeMultiTypeDataServer.C"

XIV.4.26 Macro "launchCodeMultiTypeColumn.C"

XIV.4.26.1 Objective

The objective of this macro is to test the case where vectors and strings are produced as outputs, using the code described in Section XIV.4.22.1, with a Column format, obtained by doing:

```
multitype -mtCol
```

The resulting output file, named _output_multitype_mt_Column_.dat looks like:

```
nine
-0.512095 0.039669 -1.3834 1.37667 0.220672 0.633267 1.37027 -0.765636
14.1981 14.0855 10.7848 9.45476 9.17308 6.60804 10.0711 14.1761 10.318 12.5095 15.6614 ↩
    10.3452 9.41101 7.47887
32.2723
eight
```

XIV.4.26.2 Macro Uranie

```
{
  //Create dataserver with the seed attribute
  TDataServer *tds = new TDataServer("foo","pouet");
  tds->addAttribute( new TUniformDistribution("seed",0,100000));
  tds->getAttribute("seed")->setFileKey("multitype_input.dat","", "%e",TAttributeFileKey:: ↩
      kNewRow);

  //Create DOE
  TSampling *tsam = new TSampling(tds,"lhs",100);
  tsam->generateSample();

  //Precise output and create Code
  TOutputFileColumn *out = new TOutputFileColumn("_output_multitype_mt_Column_.dat");
  out->addAttribute(new TAttribute("w1",TAttribute::kString),1);
  out->addAttribute(new TAttribute("w2",TAttribute::kString),5);
  out->addAttribute(new TAttribute("v1",TAttribute::kVector),2);
  out->addAttribute(new TAttribute("v2",TAttribute::kVector),3);
  out->addAttribute(new TAttribute("f1"),4);
  TCode *myc = new URANIE::Launcher::TCode(tds, "multitype -mtCol");
  myc->addOutputFile( out );

  //Create TLauncher and run it
  TLauncher *tlau = new URANIE::Launcher::TLauncher(tds, myc);
  tlau->run();

  //Produce control plot
  TCanvas *Can = new TCanvas("Can","Can",10,10,1000,800);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw(); pad->cd();
  tds->drawPairs("w1:v1:v2:f1:w2");

}
```

When looking at the code and comparing it to an usual launcher job, the organisation is completely transparent. A dataserver is created with a simple input (the seed) drawn as an uniform distribution. The design-of-experiments is generated and the `TOutputFileColumn` is created and filled with new attributes. This is the important part, where nature of the attributes are precised:

```
TOutputFileColumn *out = new TOutputFileColumn("_output_multitype_mt_Column_.dat");
out->addAttribute(new URANIE::DataServer::TAttribute("w1",TAttribute::kString),1);
out->addAttribute(new URANIE::DataServer::TAttribute("w2",TAttribute::kString),5);
out->addAttribute(new URANIE::DataServer::TAttribute("v1",TAttribute::kVector),2);
out->addAttribute(new URANIE::DataServer::TAttribute("v2",TAttribute::kVector),3);
out->addAttribute(new URANIE::DataServer::TAttribute("f1"),4);
```

The rest is very common and a pair plot is performed to check that the reading of the output went well (see following plot).

XIV.4.26.3 Graph

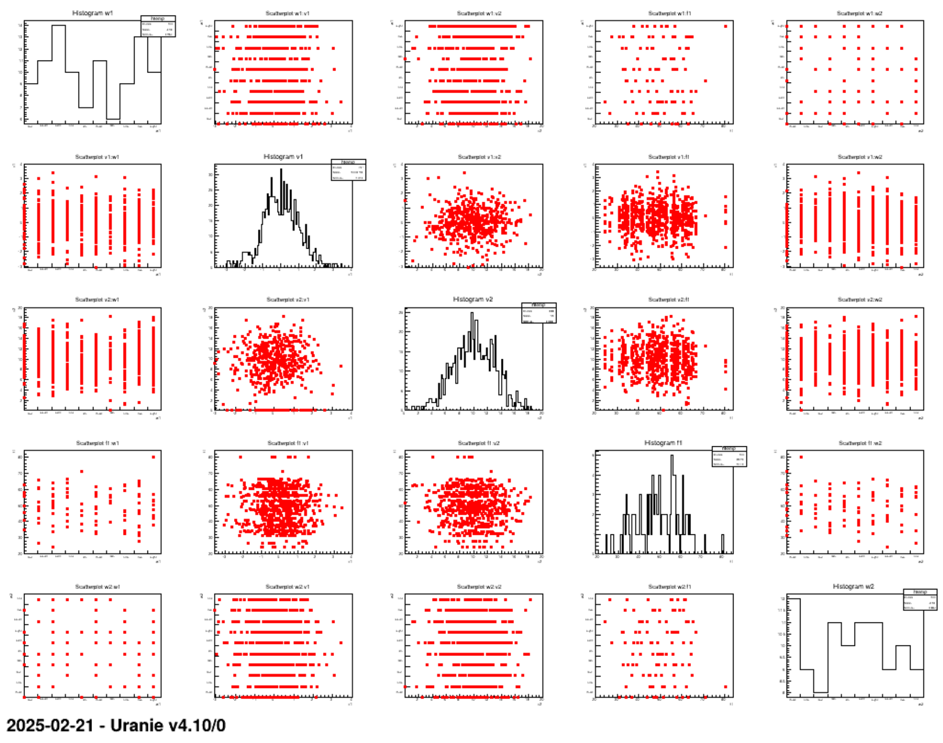


Figure XIV.41: Graph of the macro "`launchCodeMultiTypeColumn.C`"

XIV.4.27 Macro "`launchCodeMultiTypeRow.C`"

XIV.4.27.1 Objective

The objective of this macro is to test the case where vectors and strings are produced as outputs, using the code described in Section XIV.4.22.1, with a Row format, obtained by doing:

```
multitype -mtRow
```

When requesting a "row" type output, extra caution has to be taken: the usual separator between two fields are blank and/or tabulation. With this formatting, an output can look like this:

```
1.234  4.321  5.653
5.321
```

With this kind of file, it is impossible to know which attributes own the element on the second line (in other words, if it should have been, first, second or third column). The proposed solution is to change the separator between field, using any specific sign, followed by a blank. This is done using ";" and it results in an output file, named `_output_multitype_mt_Row_.dat` which looks like:

```
nine; -0.512095; 14.1981; 32.2723; eight
; 0.039669; 14.0855; ; ;
; -1.3834; 10.7848; ; ;
; 1.37667; 9.45476; ; ;
; 0.220672; 9.17308; ; ;
; 0.633267; 6.60804; ; ;
; 1.37027; 10.0711; ; ;
; -0.765636; 14.1761; ; ;
; ; 10.318; ; ;
; ; 12.5095; ; ;
; ; 15.6614; ; ;
; ; 10.3452; ; ;
; ; 9.41101; ; ;
; ; 7.47887; ; ;
```

XIV.4.27.2 Macro Uranie

```
{
  //Create dataserver with the seed attribute
  TDataServer *tds = new TDataServer("foo","pouet");
  tds->addAttribute( new TUniformDistribution("seed",0,100000));
  tds->getAttribute("seed")->setFileKey("multitype_input.dat","", "%e",TAttributeFileKey:: ↩
      kNewRow);

  //Create DOE
  TSampling *tsam = new TSampling(tds,"lhs",100);
  tsam->generateSample();

  //Precise output and create Code
  TOutputFileRow *out = new TOutputFileRow("_output_multitype_mt_Row_.dat");
  out->setFieldSeparatorCharacter(";");
  out->addAttribute(new TAttribute("w1",TAttribute::kString));
  out->addAttribute(new TAttribute("v1",TAttribute::kVector));
  out->addAttribute(new TAttribute("v2",TAttribute::kVector));
  out->addAttribute(new TAttribute("f1"));
  out->addAttribute(new TAttribute("w2",TAttribute::kString));
  TCode *myc = new URANIE::Launcher::TCode(tds, "multitype -mtRow");
  myc->addOutputFile( out );

  //Create TLauncher and run it
  TLauncher *tlau = new URANIE::Launcher::TLauncher(tds, myc);
  tlau->run();

  //Produce control plot
  TCanvas *Can = new TCanvas("Can","Can",10,10,1000,800);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw(); pad->cd();
  tds->drawPairs("w1:v1:v2:f1:w2");

}
```

When looking at the code and comparing it to an usual launcher job, the organisation is completely transparent. A dataserver is created with a simple input (the seed) drawn as an uniform distribution. The design-of-experiments is generated and the `TOutputFileRow` is created and filled with new attributes. This is the important part, where nature of the attributes are precised:

```
TOutputFileRow *out = new TOutputFileRow("_output_multitype_mt_Row_.dat");
out->setFieldSeparatorCharacter(";");
out->addAttribute(new URANIE::DataServer::TAttribute("w1",TAttribute::kString));
out->addAttribute(new URANIE::DataServer::TAttribute("v1",TAttribute::kVector));
out->addAttribute(new URANIE::DataServer::TAttribute("v2",TAttribute::kVector));
out->addAttribute(new URANIE::DataServer::TAttribute("f1"));
out->addAttribute(new URANIE::DataServer::TAttribute("w2",TAttribute::kString));
```

The second line is important as it explains the `TOutputFileRow` object how to read the file to know that some fields are empty. The rest is very common and a pair plot is performed to check that the reading of the output went well (see following plot).

XIV.4.27.3 Graph

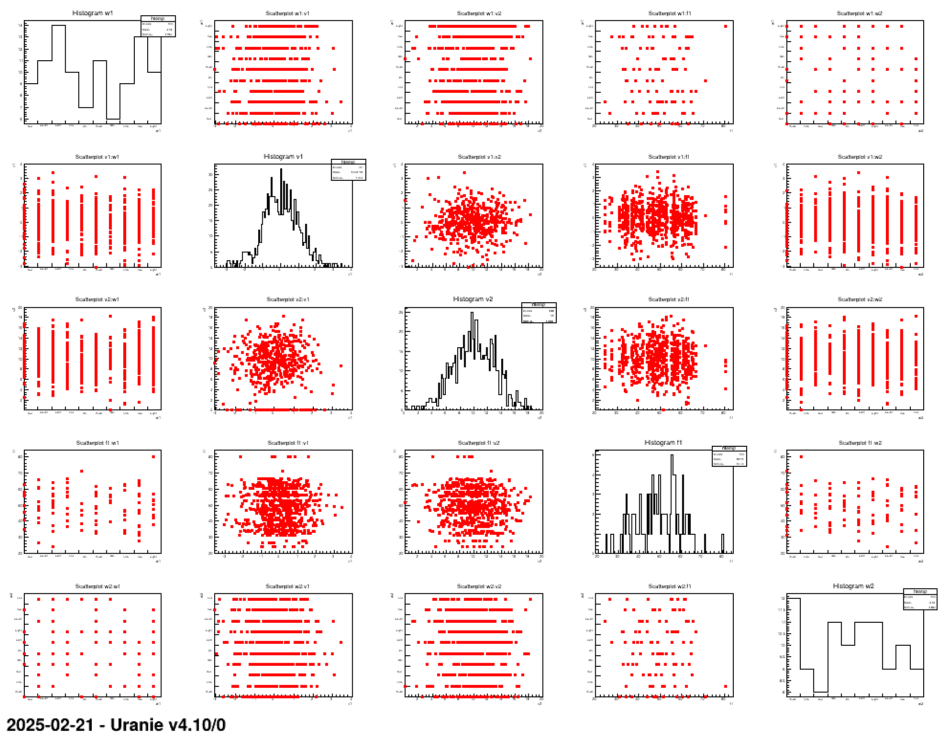


Figure XIV.42: Graph of the macro "`launchCodeMultiTypeRow.C`"

XIV.4.28 Macro "`launchCodeMultiTypeXML.C`"

XIV.4.28.1 Objective

The objective of this macro is to test the case where vectors and strings are produced as outputs, using the code described in Section XIV.4.22.1, with a XML format, obtained by doing:

```
multitype -mtXML
```

The resulting output file is named `_output_multitype_mt_.xml` and looks like:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<multitypeMT>
 <w1 value="nine"/>
 <v1 n="8">
  <values>
    -0.512095 0.039669 -1.3834 1.37667 0.220672 0.633267 1.37027 -0.765636
  </values>
 </v1>
 <v2 n="14">
  <values>
```

```
    14.1981 14.0855 10.7848 9.45476 9.17308 6.60804 10.0711 14.1761 10.318 12.5095 15.6614  ↩
        10.3452 9.41101 7.47887
  </values>
 </v2>
 <f1 value="32.2723"/>
 <w2 value="eight"/>
</multitypeMT>
```

XIV.4.28.2 Macro Uranie

```
{
  //Create dataserver with the seed attribute
  TDataServer *tds = new TDataServer("foo","pouet");
  tds->addAttribute( new TUniformDistribution("seed",0,100000));
  tds->getAttribute("seed")->setFileKey("multitype_input.dat","", "%e",TAttributeFileKey:: ↩
      kNewRow);

  //Create DOE
  TSampling *tsam = new TSampling(tds,"lhs",100);
  tsam->generateSample();

  //Precise output and create Code
  TOutputFileXML *out = new TOutputFileXML("_output_multitype_mt_.xml");
  out->addAttribute(new TAttribute("w1",TAttribute::kString), "/multitypeMT/w1/@value",  ↩
      TAttributeFileKey::kXMLAttribute );
  out->addAttribute(new TAttribute("w2",TAttribute::kString), "/multitypeMT/w2/@value",  ↩
      TAttributeFileKey::kXMLAttribute );
  out->addAttribute(new TAttribute("v1",TAttribute::kVector), "/multitypeMT/v1/values",  ↩
      TAttributeFileKey::kXMLField );
  out->addAttribute(new TAttribute("v2",TAttribute::kVector), "/multitypeMT/v2/values",  ↩
      TAttributeFileKey::kXMLField );
  out->addAttribute(new TAttribute("f1"),                     "/multitypeMT/f1/@value",  ↩
      TAttributeFileKey::kXMLAttribute );

  TCode *myc = new URANIE::Launcher::TCode(tds, "multitype -mtXML");
  myc->addOutputFile( out );

  //Create TLauncher and run it
  TLauncher *tlau = new URANIE::Launcher::TLauncher(tds, myc);
  tlau->run();

  //Produce control plot
  TCanvas *Can = new TCanvas("Can","Can",10,10,1000,800);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw(); pad->cd();
  tds->drawPairs("w1:v1:v2:f1:w2");

}
```

When looking at the code and comparing it to an usual launcher job, the organisation is completely transparent. A dataserver is created with a simple input (the seed) drawn as an uniform distribution. The design-of-experiments is generated and the `TOutputFileXML` is created and filled with new attributes. This is the important part, where nature of the attributes are precised:

```
TOutputFileXML *out = new TOutputFileXML("_output_multitype_mt_.xml");
out->addAttribute(new TAttribute("w1",TAttribute::kString), "/multitypeMT/w1/@value",  ↩
    TAttributeFileKey::kXMLAttribute );
out->addAttribute(new TAttribute("w2",TAttribute::kString), "/multitypeMT/w2/@value",  ↩
    TAttributeFileKey::kXMLAttribute );
```

```
out->addAttribute(new TAttribute("v1",TAttribute::kVector), "/multitypeMT/v1/values",  ↩
    TAttributeFileKey::kXMLField );
out->addAttribute(new TAttribute("v2",TAttribute::kVector), "/multitypeMT/v2/values",  ↩
    TAttributeFileKey::kXMLField );
out->addAttribute(new TAttribute("f1"),                     "/multitypeMT/f1/@value",  ↩
    TAttributeFileKey::kXMLAttribute );
```

The second line is important as it explains the `TOutputFileXML` object how to read the file to know that some fields are empty. The rest is very common and a pair plot is performed to check that the reading of the output went well (see following plot).

XIV.4.28.3 Graph

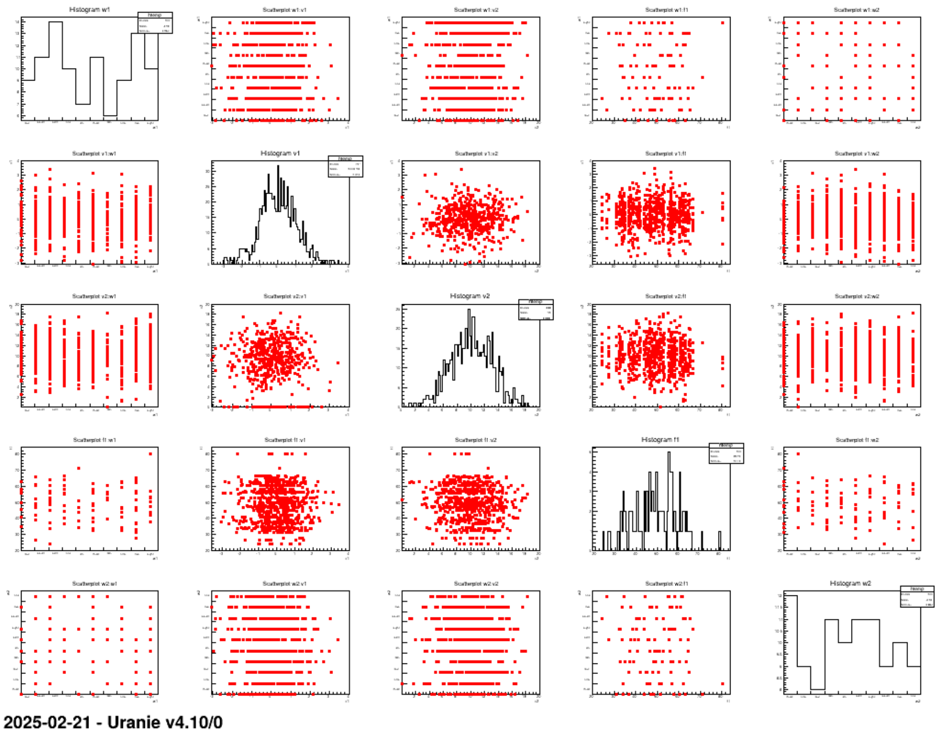


Figure XIV.43: Graph of the macro "`launchCodeMultiTypeXML.C`"

XIV.4.29 Macro "`launchCodeReadMultiTypeKey.C`"

XIV.4.29.1 Objective

The objective of this macro is to test the case where vectors and strings are used as inputs, using the code described in Section XIV.4.22.2, with a Key format, obtained by doing:

```
multitype -ReadmtKey
```

The input values will be read from a database which is produced with the `multitype -mt` code, as no sampling is available yet to produce vectors and strings. The database file is `readmultitype_sampling.dat` which looks like this:

```
#NAME: foo
#TITLE: TDS for flowrate
#DATE: Mon Oct  3 23:50:34 2016
#COLUMN_NAMES: v1| w1| v2| w2| f1| foo__n__iter__
#COLUMN_TYPES: V|S|V|S|D|D

-6.901933299378e-02,-1.292435959913e-01,4.558876683004e-01,5.638486368789e ↩
    -01,-4.767582766745e-02,7.102109543136e-03,2.819049677902e-01,-2.019788081790e ↩
    +00,-2.604401028584e+00,-1.617682380292e+00,2.894560949798e-02,-3.493905850261e-01 six  ↩
    1.142449011404e+01,7.318948216271e+00,1.502260859231e+01,6.041193793062e ↩
    +00,6.729445145907e+00,1.128096968597e+01 zero 3.425632316777e+01 0.000000000000e+00
-6.923200061823e-01,-4.798721931875e-01,-1.329893204384e+00,1.292933726829e+00 zero  ↩
    1.249911290435e+01,6.309239169117e+00,1.596653626442e+01,5.500878012739e ↩
    +00,1.322535550082e+01,7.070984389647e+00,1.708574150702e+00,1.265915339220e+01 two  ↩
    4.295175025115e+01 1.000000000000e+00
5.773813268848e-01,-3.512405673973e-01,-6.870089014992e-01,1.273074555211e-01 nine  ↩
    1.242682578759e+01,1.109680842701e+01,1.670410641828e+01,7.296321908492e ↩
    +00,8.732800753443e+00,1.262906549132e+01,8.882310687564e+00,1.104280818003e+01 five  ↩
    5.591437936893e+01 2.000000000000e+00
5.518508915499e-01,2.438158138873e-01,1.111784497742e+00,-1.517566514667e+00,7.146879916125 ↩
    e-01,2.328439269321e+00,-1.251913839951e+00,8.876684186954e-01,-1.383023165632e ↩
    +00,-8.192089693621e-01,-1.079524713568e-01,6.595650273375e-01,-2.275345802432e ↩
    -03,1.304354557600e+00 nine 1.021975159505e+01,4.995433740783e+00,1.108628156181e ↩
    +01,1.041110604995e+01,1.111365770153e+01,6.365695806343e+00,6.374053973239e ↩
    +00,6.854423942510e+00,7.144262333164e+00 two 4.093776591421e+01 3.000000000000e+00
2.403942476958e-01,6.868091212609e-01,-1.561012830108e+00,1.937806684989e ↩
    +00,-1.465851888061e+00,5.367279844359e-02,-1.263005327899e+00,-1.132259472701e+00 two  ↩
    7.382048319627e+00,5.874867917970e+00,1.158191378461e+01,1.073321314846e+01 six  ↩
    6.980549752305e+01 4.000000000000e+00
2.220485143391e+00,-5.787212569267e-01,8.843648237689e-01,2.020662891124e+00,1.066403357312 ↩
    e+00,-5.817432767992e-01,3.063023900800e-01,-7.393588637933e-01 two 2.049656723853e ↩
    +00,9.679003878866e+00,7.338089623518e+00,1.235630702472e+01,1.509238505697e ↩
    +01,1.034077492413e+01,1.116077550501e+01,7.179221834787e+00,1.582041236432e ↩
    +01,9.204085091129e+00,4.707490792498e+00,1.618155764288e+01 five 3.507773555061e+01  ↩
    5.000000000000e+00
8.908373817765e-01,-2.446355046704e-01,-1.900125532005e+00 seven 1.351254851860e ↩
    +01,9.297087139459e+00,1.130966904782e+01,1.219245848701e+01,1.012996566249e ↩
    +01,7.150071600452e+00,1.097549218518e+01,1.443074761657e+01 five 4.464560504112e+01  ↩
    6.000000000000e+00
-2.514644600888e+00,1.633579305804e+00 one 1.229098312451e+01,1.013486836958e ↩
    +01,1.243386772880e+01,1.071783135260e+01,1.453735777922e+01,7.995593455015e ↩
    +00,9.753966962919e+00,5.924583770352e+00,6.187713988125e+00,1.061975242996e ↩
    +01,6.650425922126e+00 four 4.553396475968e+01 7.000000000000e+00
-1.347811599520e+00,-1.259450135534e+00,1.812553405758e+00 five 7.717018655412e ↩
    +00,1.053283796180e+01,7.404059210327e+00 eight 6.695868880279e+01 8.000000000000e+00
-1.258360863204e-01,-9.000566818602e-01,7.039146852797e-01,1.015917277706e ↩
    +00,-2.397650482929e-01 four 4.346717386417e+00,1.033024889324e+01,7.183787459050e ↩
    +00,8.742095837835e+00,1.277095440277e+01,8.685683828779e+00,9.321006265935e ↩
    +00,6.353438157123e+00,8.552570119034e+00 six 4.381313066586e+01 9.000000000000e+00
```

For every pattern, an input file is created with the Key condensate format, as the other key format is not practical (and usable). This input file looks like this:

```
w1 = nine
v1 = [ -0.512095,0.039669,-1.3834,1.37667,0.220672,0.633267,1.37027,-0.765636 ]
v2 = [  ↩
    14.1981,14.0855,10.7848,9.45476,9.17308,6.60804,10.0711,14.1761,10.318,12.5095,15.6614,10.3452,9.41101
     ]
f1 = 32.2723
w2 = eight
```

The resulting output file, named `_output_multitype_readmt_Key_.dat` looks like:

```
thev1 = -0.2397650482929
thev2 = 9.321006265935
```

XIV.4.29.2 Macro Uranie

```
{
  //Create dataserver with a database of 10 runs of the "multitype -mt" code
  TDataServer *tds = new TDataServer("foo","pouet");
  tds->fileDataRead("readmultitype_sampling.dat");

  //Explain the name of the input file and the format chosen
  tds->getAttribute("w1")->setFileKey("_output_multitype_mt_Key_condensate_.dat","", "%e", ↩
      TAttributeFileKey::kNewKey);
  tds->getAttribute("v1")->setFileKey("_output_multitype_mt_Key_condensate_.dat","", "%e", ↩
      TAttributeFileKey::kNewKey);
  tds->getAttribute("v2")->setFileKey("_output_multitype_mt_Key_condensate_.dat","", "%e", ↩
      TAttributeFileKey::kNewKey);
  tds->getAttribute("f1")->setFileKey("_output_multitype_mt_Key_condensate_.dat","", "%e", ↩
      TAttributeFileKey::kNewKey);
  tds->getAttribute("w2")->setFileKey("_output_multitype_mt_Key_condensate_.dat","", "%e", ↩
      TAttributeFileKey::kNewKey);

  //Create the Class that will handle re-writting the input file
  TString OutName="_output_multitype_mt_Key_condensate_.dat";
  TInputFileRecreate *in = new TInputFileRecreate(gSystem->PrependPathName(gSystem->pwd(), ↩
      OutName));

  // Add attribute in the correct (needed) order
  in->addAttribute( tds->getAttribute("w1") );
  in->addAttribute( tds->getAttribute("v1") );
  in->addAttribute( tds->getAttribute("v2") );
  in->addAttribute( tds->getAttribute("f1") );
  in->addAttribute( tds->getAttribute("w2") );
  in->setVectorProperties("[",",","]"); // Change the vector properties

  // Create the output file interface and state that there will be 2 outputs
  TOutputFileKey *out = new TOutputFileKey("_output_multitype_readmt_Key_.dat");
  out->addAttribute(new TAttribute("thev1") );
  out->addAttribute(new TAttribute("thev2") );

  // Create the corresponding interface to code
  TCode *tc1 = new TCode(tds, "multitype -ReadmtKey");
  tc1->addInputFile( in );
  tc1->addOutputFile( out );

  // Create the launcher and run the code
  TLauncher *tl1 = new TLauncher(tds, tc1);
  tl1->run();

  tds->Scan("thev1:thev2");

}
```

When looking at the code and comparing it to an usual launcher job, the organisation is completely transparent. A dataserver is created by reading an input file `readmultitype_sampling.dat` in which 10 events are stored. The main difference arises from the way the input file is created: it is done explicitly because the order in which the attributes are stored in the database file are not the one needed by the code in the input file (this case is discussed in

Section IV.3.1.1). The name of the file given in the construction has to be absolute, as the `TCode` object will check this in order to know whether the input file should be created.

```
//Create the Class that will handle re-writing the input file
TInputFileRecreate *in = new TInputFileRecreate(Form("%s/ ↩
    _output_multitype_mt_Key_condensate_.dat",gSystem->pwd()));

// Add attribute in the correct (needed) order
in->addAttribute( tds->getAttribute("w1") );
in->addAttribute( tds->getAttribute("v1") );
in->addAttribute( tds->getAttribute("v2") );
in->addAttribute( tds->getAttribute("f1") );
in->addAttribute( tds->getAttribute("w2") );
in->setVectorProperties("[",",",","]"); // Change the vector properties
```

Moreover, an extra line is added to specify that the Key format is a condensate one, so that Uranie knows how to deal with vectors and strings specific case (only vectors here). The rest is very common and a screenshot of the result displayed in console is provided in the following subsection.

XIV.4.29.3 Console

```
Processing launchCodeReadMultiTypeKey.C...
************************************
*    Row   *     thev1 *     thev2 *
************************************
*        0 * 0.2819049 * 11.424490 *
*        1 * -0.692320 * 15.966536 *
*        2 * -12345678 * 12.629065 *
*        3 * -0.819208 * 11.086281 *
*        4 * -1.561012 * -12345678 *
*        5 * 0.8843648 * 10.340774 *
*        6 * -12345678 * 7.1500716 *
*        7 * 1.6335793 * 14.537357 *
*        8 * -12345678 * -12345678 *
*        9 * -0.239765 * 9.3210062 *
************************************
```

XIV.4.30 Macro "launchCodeReadMultiTypeDataServer.C"

XIV.4.30.1 Objective

The objective of this macro is to test the case where vectors and strings are used as inputs, using the code described in Section XIV.4.22.2, with a DataServer format, obtained by doing:

```
multitype -ReadmtDS
```

. The input values will be read from a database which is produced with the `multitype -mt` code, as no sampling is available yet to produce vectors and strings. The database file is `readmultitype_sampling.dat` which is shown in Section XIV.4.29.1. For every pattern, an input file will be created with the DataServer format. This input file looks like this:

```
#COLUMN_NAMES: w1|v1|v2|f1|w2
#COLUMN_TYPES: S|V|V|D|S

nine -0.512095,0.039669,-1.3834,1.37667,0.220672,0.633267,1.37027,-0.765636  ↩
    14.1981,14.0855,10.7848,9.45476,9.17308,6.60804,10.0711,14.1761,10.318,12.5095,15.6614,10.3452,9.41101
     32.2723 eight
```

The resulting output file, named `_output_multitype_readmt_DataServer_.dat` looks like:

```
#COLUMN_NAMES: thev1|thev2
#COLUMN_TYPES: D|D

-0.2397650482929 9.321006265935
```

XIV.4.30.2 Macro Uranie

```
{
  //Create dataserver with a database of 10 runs of the "multitype -mt" code
  TDataServer *tds = new TDataServer("foo","pouet");
  tds->fileDataRead("readmultitype_sampling.dat");

  //Explain the name of the input file and the format chosen
  tds->getAttribute("w1")->setFileKey("_output_multitype_mt_DataServer_.dat","", "%e", ↩
      TAttributeFileKey::kNewTDS);
  tds->getAttribute("v1")->setFileKey("_output_multitype_mt_DataServer_.dat","", "%e", ↩
      TAttributeFileKey::kNewTDS);
  tds->getAttribute("v2")->setFileKey("_output_multitype_mt_DataServer_.dat","", "%e", ↩
      TAttributeFileKey::kNewTDS);
  tds->getAttribute("f1")->setFileKey("_output_multitype_mt_DataServer_.dat","", "%e", ↩
      TAttributeFileKey::kNewTDS);
  tds->getAttribute("w2")->setFileKey("_output_multitype_mt_DataServer_.dat","", "%e", ↩
      TAttributeFileKey::kNewTDS);

  //Create the Class that will handle re-writting the input file
  TString OutName="_output_multitype_mt_DataServer_.dat";
  TInputFileRecreate *in = new TInputFileRecreate(gSystem->PrependPathName(gSystem->pwd(), ↩
      OutName));

  // Add attribute in the correct (needed) order
  in->addAttribute( tds->getAttribute("w1") );
  in->addAttribute( tds->getAttribute("v1") );
  in->addAttribute( tds->getAttribute("v2") );
  in->addAttribute( tds->getAttribute("f1") );
  in->addAttribute( tds->getAttribute("w2") );

  // Create the output file interface and state that there will be 2 outputs
  TOutputFileDataServer *out = new TOutputFileDataServer(" ↩
      _output_multitype_readmt_DataServer_.dat");
  out->addAttribute(new TAttribute("thev1") );
  out->addAttribute(new TAttribute("thev2") );

  // Create the corresponding interface to code
  TCode *tc1 = new TCode(tds, "multitype -ReadmtDS");
  tc1->addInputFile( in );
  tc1->addOutputFile( out );

  // Create the launcher and run the code
  TLauncher *tl1 = new TLauncher(tds, tc1);
  tl1->run();

  tds->Scan("thev1:thev2");

}
```

When looking at the code and comparing it to an usual launcher job, the organisation is completely transparent. A dataserver is created by reading an input file `readmultitype_sampling.dat` in which 10 events are stored.

The main difference arises from the way the input file is created: it is done explicitly because the order in which the attributes are stored in the database file are not the one needed by the code in the input file (this case is discussed in Section IV.3.1.1). The name of the file given in the construction has to be absolute, as the `TCode` object will check this in order to know whether the input file should be created.

```
//Create the Class that will handle re-writing the input file
TInputFileRecreate *in = new TInputFileRecreate(Form("%s/_output_multitype_mt_DataServer_. ↩
    dat",gSystem->pwd()));

// Add attribute in the correct (needed) order
in->addAttribute( tds->getAttribute("w1") );
in->addAttribute( tds->getAttribute("v1") );
in->addAttribute( tds->getAttribute("v2") );
in->addAttribute( tds->getAttribute("f1") );
in->addAttribute( tds->getAttribute("w2") );
```

The rest is very common and a screenshot of the result displayed in console is provided in the following subsections.

XIV.4.30.3 Console

```
Processing launchCodeReadMultiTypeDataServer.C...
************************************
*    Row   *     thev1 *     thev2 *
************************************
*        0 * 0.2819049 * 11.424490 *
*        1 * -0.692320 * 15.966536 *
*        2 * -12345678 * 12.629065 *
*        3 * -0.819208 * 11.086281 *
*        4 * -1.561012 * -12345678 *
*        5 * 0.8843648 * 10.340774 *
*        6 * -12345678 * 7.1500716 *
*        7 * 1.6335793 * 14.537357 *
*        8 * -12345678 * -12345678 *
*        9 * -0.239765 * 9.3210062 *
************************************
```

XIV.4.31 Macro "`launchCodeReadMultiTypeColumn.C`"

XIV.4.31.1 Objective

The objective of this macro is to test the case where vectors and strings are used as inputs, using the code described in Section XIV.4.22.2, with a Column format, obtained by doing:

```
multitype -ReadmtCol
```

The input values will be read from a database which is produced with the `multitype -mt` code, as no sampling is available yet to produce vectors and strings. The database file is `readmultitype_sampling.dat` which is shown in Section XIV.4.29.1. For every pattern, an input file will be created with the Column format. This input file looks like this:

```
nine
-0.512095 0.039669 -1.3834 1.37667 0.220672 0.633267 1.37027 -0.765636
14.1981 14.0855 10.7848 9.45476 9.17308 6.60804 10.0711 14.1761 10.318 12.5095 15.6614  ↩
    10.3452 9.41101 7.47887
32.2723
eight
```

The resulting output file, named `_output_multitype_readmt_Column_.dat` looks like:

```
-0.2397650482929
9.321006265935
```

XIV.4.31.2 Macro Uranie

```
{
  //Create dataserver with a database of 10 runs of the "multitype -mt" code
  TDataServer *tds = new TDataServer("foo","pouet");
  tds->fileDataRead("readmultitype_sampling.dat");

  //Explain the name of the input file and the format chosen
  tds->getAttribute("w1")->setFileKey("_output_multitype_mt_Column_.dat","", "%e", ↩
      TAttributeFileKey::kNewColumn);
  tds->getAttribute("v1")->setFileKey("_output_multitype_mt_Column_.dat","", "%e", ↩
      TAttributeFileKey::kNewColumn);
  tds->getAttribute("v2")->setFileKey("_output_multitype_mt_Column_.dat","", "%e", ↩
      TAttributeFileKey::kNewColumn);
  tds->getAttribute("f1")->setFileKey("_output_multitype_mt_Column_.dat","", "%e", ↩
      TAttributeFileKey::kNewColumn);
  tds->getAttribute("w2")->setFileKey("_output_multitype_mt_Column_.dat","", "%e", ↩
      TAttributeFileKey::kNewColumn);

  //Create the Class that will handle re-writting the input file
  TString OutName="_output_multitype_mt_Column_.dat";
  TInputFileRecreate *in = new TInputFileRecreate(gSystem->PrependPathName(gSystem->pwd(), ↩
      OutName));

  // Add attribute in the correct (needed) order
  in->addAttribute( tds->getAttribute("w1") );
  in->addAttribute( tds->getAttribute("v1") );
  in->addAttribute( tds->getAttribute("v2") );
  in->addAttribute( tds->getAttribute("f1") );
  in->addAttribute( tds->getAttribute("w2") );

  // Create the output file interface and state that there will be 2 outputs
  TOutputFileColumn *out = new TOutputFileColumn("_output_multitype_readmt_Column_.dat");
  out->addAttribute(new TAttribute("thev1") );
  out->addAttribute(new TAttribute("thev2") );

  // Create the corresponding interface to code
  TCode *tc1 = new TCode(tds, "multitype -ReadmtCol");
  tc1->addInputFile( in );
  tc1->addOutputFile( out );

  // Create the launcher and run the code
  TLauncher *tl1 = new TLauncher(tds, tc1);
  tl1->run();

  tds->Scan("thev1:thev2");

}
```

When looking at the code and comparing it to an usual launcher job, the organisation is completely transparent. A dataserver is created by reading an input file `readmultitype_sampling.dat` in which 10 events are stored. The main difference arises from the way the input file is created: it is done explicitly because the order in which the attributes are stored in the database file are not the one needed by the code in the input file (this case is discussed in

Section IV.3.1.1). The name of the file given in the construction has to be absolute, as the `TCode` object will check this in order to know whether the input file should be created.

```
//Create the Class that will handle re-writing the input file
TInputFileRecreate *in = new TInputFileRecreate(Form("%s/_output_multitype_mt_Column_.dat", ↩
    gSystem->pwd()));

// Add attribute in the correct (needed) order
in->addAttribute( tds->getAttribute("w1") );
in->addAttribute( tds->getAttribute("v1") );
in->addAttribute( tds->getAttribute("v2") );
in->addAttribute( tds->getAttribute("f1") );
in->addAttribute( tds->getAttribute("w2") );
```

The rest is very common and a screenshot of the result displayed in console is provided in the following subsection.

XIV.4.31.3 Console

```
Processing launchCodeReadMultiTypeColumn.C...
************************************
*    Row   *     thev1 *     thev2 *
************************************
*        0 * 0.2819049 * 11.424490 *
*        1 * -0.692320 * 15.966536 *
*        2 * -12345678 * 12.629065 *
*        3 * -0.819208 * 11.086281 *
*        4 * -1.561012 * -12345678 *
*        5 * 0.8843648 * 10.340774 *
*        6 * -12345678 * 7.1500716 *
*        7 * 1.6335793 * 14.537357 *
*        8 * -12345678 * -12345678 *
*        9 * -0.239765 * 9.3210062 *
************************************
```

XIV.4.32 Macro "`launchCodeReadMultiTypeRow.C`"

XIV.4.32.1 Objective

The objective of this macro is to test the case where vectors and strings are used as inputs, using the code described in Section XIV.4.22.2, with a Row format, obtained by doing:

```
multitype -ReadmtRow
```

The input values will be read from a database which has been produced with the `multitype -mt` code, as no sampling is available yet to produce vectors and strings. The database file is `readmultitype_sampling.dat` which is shown in Section XIV.4.29.1. For every pattern, an input file will be created with the Row format. This input file is very peculiar as the number of entries per attribute is not equal going from attribute to another (this format is not recommended as discussed in the third item of Section IV.3.1.2.3). It will look like this:

```
nine; -0.512095; 14.1981; 32.2723; eight
; 0.039669; 14.0855; ; ;
; -1.3834; 10.7848; ; ;
; 1.37667; 9.45476; ; ;
; 0.220672; 9.17308; ; ;
; 0.633267; 6.60804; ; ;
; 1.37027; 10.0711; ; ;
; -0.765636; 14.1761; ; ;
; ; 10.318; ; ;
; ; 12.5095; ; ;
; ; 15.6614; ; ;
; ; 10.3452; ; ;
; ; 9.41101; ; ;
; ; 7.47887; ; ;
```

The resulting output file, named `_output_multitype_readmt_Row_.dat`, looks like:

```
-0.2397650482929; 9.321006265935
```

XIV.4.32.2 Macro Uranie

```
{
  //Create dataserver with a database of 10 runs of the "multitype -mt" code
  TDataServer *tds = new TDataServer("foo","pouet");
  tds->fileDataRead("readmultitype_sampling.dat");

  //Explain the name of the input file and the format chosen
  tds->getAttribute("w1")->setFileKey("_output_multitype_mt_Row_.dat","", "%e", ↩
      TAttributeFileKey::kNewRow);
  tds->getAttribute("v1")->setFileKey("_output_multitype_mt_Row_.dat","", "%e", ↩
      TAttributeFileKey::kNewRow);
  tds->getAttribute("v2")->setFileKey("_output_multitype_mt_Row_.dat","", "%e", ↩
      TAttributeFileKey::kNewRow);
  tds->getAttribute("f1")->setFileKey("_output_multitype_mt_Row_.dat","", "%e", ↩
      TAttributeFileKey::kNewRow);
  tds->getAttribute("w2")->setFileKey("_output_multitype_mt_Row_.dat","", "%e", ↩
      TAttributeFileKey::kNewRow);

  //Create the Class that will handle re-writting the input file
  TString OutName="_output_multitype_mt_Row_.dat";
  TInputFileRecreate *in = new TInputFileRecreate( gSystem->PrependPathName(gSystem->pwd(), ↩
      OutName));

  // Add attribute in the correct (needed) order
  in->addAttribute( tds->getAttribute("w1") );
  in->addAttribute( tds->getAttribute("v1") );
  in->addAttribute( tds->getAttribute("v2") );
  in->addAttribute( tds->getAttribute("f1") );
  in->addAttribute( tds->getAttribute("w2") );
  in->setFieldSeparatorCharacter("; "); // Change the separator

  // Create the output file interface and state that there will be 2 outputs
  TOutputFileRow *out = new TOutputFileRow("_output_multitype_readmt_Row_.dat");
  out->addAttribute(new TAttribute("thev1") );
  out->addAttribute(new TAttribute("thev2") );

  // Create the corresponding interface to code
  TCode *tc1 = new TCode(tds, "multitype -ReadmtRow");
  tc1->addInputFile( in );
  tc1->addOutputFile( out );

  // Create the launcher and run the code
  TLauncher *tl1 = new TLauncher(tds, tc1);
  tl1->run();

  tds->Scan("thev1:thev2");
}
```

When looking at the code and comparing it to an usual launcher job, the organisation is completely transparent. A dataserver is created by reading an input file `readmultitype_sampling.dat` in which 10 events are stored. The main difference arises from the way the input file is created: it is done explicitly because the order in which the attributes are stored in the database file are not the one needed by the code in the input file (this case is discussed in Section IV.3.1.1). The name of the file given in the construction has to be absolute, as the `TCode` object will check this in order to know whether the input file should be created.

```
//Create the Class that will handle re-writing the input file
TInputFileRecreate *in = new TInputFileRecreate(Form("%s/_output_multitype_mt_Row_.dat", ↩
    gSystem->pwd()));

// Add attribute in the correct (needed) order
in->addAttribute( tds->getAttribute("w1") );
in->addAttribute( tds->getAttribute("v1") );
in->addAttribute( tds->getAttribute("v2") );
in->addAttribute( tds->getAttribute("f1") );
in->addAttribute( tds->getAttribute("w2") );
in->setFieldSeparatorCharacter("; "); // Change the separator
```

When requesting a "Row" type input, extra caution has to be taken: the usual separator between two fields are blank and/or tabulation. With this formatting, an input can look like this:

```
1.234  4.321  5.653
5.321
```

With this kind of file, it is impossible to know which attributes own the element on the second line (in other words, if it should have been, first, second or third column). The proposed solution is to change the separator between field, using any specific sign, followed by a blank. This is done using ";" and it is shown as the last line in previous code. The rest is very common and a screenshot of the result displayed in console is provided in the following subsection.

XIV.4.32.3 Console

```
Processing launchCodeReadMultiTypeRow.C...
************************************
*    Row   *     thev1 *     thev2 *
************************************
*        0 * 0.2819049 * 11.424490 *
*        1 * -0.692320 * 15.966536 *
*        2 * -12345678 * 12.629065 *
*        3 * -0.819208 * 11.086281 *
*        4 * -1.561012 * -12345678 *
*        5 * 0.8843648 * 10.340774 *
*        6 * -12345678 * 7.1500716 *
*        7 * 1.6335793 * 14.537357 *
*        8 * -12345678 * -12345678 *
*        9 * -0.239765 * 9.3210062 *
************************************
```

XIV.4.33 Macro "launchCodeReadMultiTypeXML.C"

XIV.4.33.1 Objective

The objective of this macro is to test the case where vectors and strings are used as inputs, using the code described in Section XIV.4.22.2, with a XML format, obtained by doing:

```
multitype -ReadmtXML
```

The input values will be read from a database which is produced with the `multitype -mt` code, as no sampling is available yet to produce vectors and strings. The database file is `readmultitype_sampling.dat` which is shown in Section XIV.4.29.1. For every pattern, an input file will be created with the XML format. This input file looks like this:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<multitypeMT>
 <w1 value="nine"/>
 <v1 n="8">
  <values>
    -0.512095 0.039669 -1.3834 1.37667 0.220672 0.633267 1.37027 -0.765636
  </values>
 </v1>
 <v2 n="14">
  <values>
    14.1981 14.0855 10.7848 9.45476 9.17308 6.60804 10.0711 14.1761 10.318 12.5095 15.6614  ↩
        10.3452 9.41101 7.47887
  </values>
 </v2>
 <f1 value="32.2723"/>
 <w2 value="eight"/>
</multitypeMT>
```

The resulting output file is named `_output_multitype_readmt_.xml` and looks like:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<multitypeREADMT>
 <thev1 value="-0.2397650482929"/>
 <thev2 value="9.321006265935"/>
</multitypeREADMT>
```

XIV.4.33.2 Macro Uranie

```
{
  //Create dataserver with a database of 10 runs of the "multitype -mt" code
  TDataServer *tds = new TDataServer("foo","pouet");
  tds->fileDataRead("readmultitype_sampling.dat");

  //Produce the input files
  gSystem->Exec("multitype -mtXML");

  //Explain the name of the input file and the format chosen
  tds->getAttribute("w1")->setFileKey("_output_multitype_mt_.xml","w1/@value", "",  ↩
      TAttributeFileKey::kXMLAttribute);
  tds->getAttribute("v1")->setFileKey("_output_multitype_mt_.xml","v1/values", "",  ↩
      TAttributeFileKey::kXMLField);
  tds->getAttribute("v2")->setFileKey("_output_multitype_mt_.xml","v2/values", "",  ↩
      TAttributeFileKey::kXMLField);
  tds->getAttribute("f1")->setFileKey("_output_multitype_mt_.xml","f1/@value", "",  ↩
      TAttributeFileKey::kXMLAttribute);
  tds->getAttribute("w2")->setFileKey("_output_multitype_mt_.xml","w2/@value", "",  ↩
      TAttributeFileKey::kXMLAttribute);

  //Create the Class that will handle re-writting the input file
  TString OutName="_output_multitype_mt_.xml";
  TInputFileXML *in = new TInputFileXML(gSystem->PrependPathName(gSystem->pwd(),OutName));

  // Add attribute in the correct (needed) order
  in->addAttribute( tds->getAttribute("w1") );
  in->addAttribute( tds->getAttribute("v1") );
  in->addAttribute( tds->getAttribute("v2") );
  in->addAttribute( tds->getAttribute("f1") );
  in->addAttribute( tds->getAttribute("w2") );

  // Create the output file interface and state that there will be 2 outputs
  TOutputFileXML *out = new TOutputFileXML("_output_multitype_readmt_.xml");
  out->addAttribute(new TAttribute("thev1"), "/flowrateREADMT/thev1/@value",   ↩
      TAttributeFileKey::kXMLAttribute);
  out->addAttribute(new TAttribute("thev2"), "/flowrateREADMT/thev2/@value",   ↩
      TAttributeFileKey::kXMLAttribute );

  // Create the corresponding interface to code
  TCode *tc1 = new TCode(tds, "multitype -ReadmtXML");
  tc1->addInputFile( in );
  tc1->addOutputFile( out );

  // Create the launcher and run the code
  TLauncher *tl1 = new TLauncher(tds, tc1);
  tl1->run();

  tds->Scan("thev1:thev2");

}
```

When looking at the code and comparing it to an usual launcher job, the organisation is completely transparent. A dataserver is created by reading an input file `readmultitype_sampling.dat` in which 10 events are stored. The main difference arises from the way the input file is created: it is done explicitly because the order in which the attributes are stored in the database file are not the one needed by the code in the input file (this case is discussed in Section IV.3.1.1). The name of the file given in the construction has to be absolute, as the `TCode` object will check this in order to know whether the input file should be created.

```
//Create the Class that will handle re-writing the input file
TInputFileXML *in = new TInputFileXML(Form("%s/_output_multitype_mt_.xml",gSystem->pwd()));

// Add attribute in the correct (needed) order
in->addAttribute( tds->getAttribute("w1") );
in->addAttribute( tds->getAttribute("v1") );
in->addAttribute( tds->getAttribute("v2") );
in->addAttribute( tds->getAttribute("f1") );
in->addAttribute( tds->getAttribute("w2") );
```

The class of input file is not a `TInputFileRecreate` as an example of input file is needed. The class used is so the `TInputFileXML` and in order to be sure that this file exists, the code is called at first as following to produce the `_output_multitype_mt_.xml`.

```
//Produce the input files
gSystem->Exec("multitype -mtXML");
```

The rest is very common and a screenshot of the result displayed in console is provided in the following subsection.

XIV.4.33.3 Console

```
Processing launchCodeReadMultiTypeXML.C...
************************************
*    Row   *      thev1 *      thev2 *
************************************
*        0 * 1.2345678 * 1.2345678 *
*        1 * 1.2345678 * 1.2345678 *
*        2 * 1.2345678 * 1.2345678 *
*        3 * 1.2345678 * 1.2345678 *
*        4 * 1.2345678 * 1.2345678 *
*        5 * 1.2345678 * 1.2345678 *
*        6 * 1.2345678 * 1.2345678 *
*        7 * 1.2345678 * 1.2345678 *
*        8 * 1.2345678 * 1.2345678 *
*        9 * 1.2345678 * 1.2345678 *
************************************
```

XIV.4.34 Macro "launchCodeFilesWithBlank.C"

XIV.4.34.1 Objective

The objective of this macro is to test the case where input and output key files are dfined with blank space in the key definition. This is a first attempt to overcome this situation and it will be using a newly-made code named `withblank` that deal with pressure and density of water in a brewery and take its input from the following file `withblank_input.dat`

```
water model.brewery pit pressure = 1E5 \\ Double
I do like Chocolate
residual brew density model.factor for glass = 1E-2 \\ Double aussi
```

The code itself is meaningless as it only multiply by two (for the pressure) and by three (for the density) and these meaningless information are stored in the output file `withblank_output.dat` which also is the form of key file, in which the keys contains several blank space, as shown below

```
Test doubling.water model.brewery pit pressure = 10.55397966411


 Bla Bla Bla


Chocolat doubling.residual brew density model.factor for glass = 2.738635531741

```

XIV.4.34.2 Macro Uranie

```
{
  // Create dataserver and add attributes
  TDataServer *tds = new TDataServer("pouet","foo");
  tds->addAttribute(new TUniformDistribution("pressure",0,10));
  tds->addAttribute(new TLogUniformDistribution("density",0.01,100));

  // Define the input file and the keys for
  TString sFileName=TString("withblank_input.dat");
  tds->getAttribute("pressure")->setFileKey(sFileName, "water model.brewery pit pressure ") ↩
      ;
  tds->getAttribute("density")->setFileKey(sFileName, "residual brew density model.factor  ↩
      for glass ");

  // Create the input file interface before it's done by the TCode to precise information
  TInputFileKey *in = new TInputFileKey(gSystem->PrependPathName(gSystem->pwd(),sFileName)) ↩
      ;
  // Set the separator to '=' so that the line could contains blank spaces (specific)
  in->setFieldSeparatorCharacter("=");
  in->setSeparatorCharacter(" ");
  in->addAttribute( tds->getAttribute("pressure") );
  in->addAttribute( tds->getAttribute("density") );

  // Generate a doe
  TBasicSampling *sampl = new TBasicSampling(tds,"srs",10);
  sampl->generateSample();

  // Define the output
  TString sOutFileName="withblank_output.dat";
  TOutputFileKey *fout = new TOutputFileKey(sOutFileName);
  // Set the separator to '=' so that the line could contains blank spaces (specific)
  fout->setFieldSeparatorCharacter("=");
  fout->setSeparatorCharacter("\n");

  // Define the attributes and their keys
  TAttribute *outAtt = new TAttribute("outpressure");
  outAtt->setFileKey(sOutFileName,"Test doubling.water model.brewery pit pressure ");
  TAttribute *outAtt2 = new TAttribute("outdensity");
  outAtt2->setFileKey(sOutFileName,"Chocolat doubling.residual brew density model.factor  ↩
      for glass ");
  fout->addAttribute(outAtt);
  fout->addAttribute(outAtt2);

  // Create a TCode object with the TDS (attribute and input files) and the command to  ↩
      execute
  TCode *mycode = new TCode(tds, "withblank");
  // Add the output file
  mycode->addInputFile( in );
  mycode->addOutputFile( fout );

  // Launcher of the code on the Design of experiments (DoE) in the TDS
  TLauncher *tlch = new TLauncher(tds, mycode);
  tlch->setSave(-1);
  tlch->setClean();
  tlch->run();

  tds->Scan("pressure:outpressure:density:outdensity");

}
```

As for all launcher's macro, it starts by defining the attributes and precise the key values with `setFileKey` method. From there, the main difference arise from the few lines below: as for the case introduced Section XIV.4.27 one needs to recreate the input file `TInputFileKey` as one needs to specify some properties

```
// Create the input file interface before it's done by the TCode to precise information
TInputFileKey *in = new TInputFileKey(gSystem->PrependPathName(gSystem->pwd(),sFileName));
// Set the separator to '=' so that the line could contains blank spaces (specific)
in->setFieldSeparatorCharacter("=");
```

```
in->setSeparatorCharacter(" ");
in->addAttribute( tds->getAttribute("pressure") );
in->addAttribute( tds->getAttribute("density") );
```

Apart from the creation of the file, the two main functions to be called are `setFieldSeparatorCharacter`, used to define the "=" character as the field delimiter while the second one `setSeparatorCharacter` cancel the usual separator cleaning procedure (usually the code is removing few characters such as ";", "\n", "="). Given this option, the file is breakdown, on a line-by-line basis, to find the field separator.

The following block is defining a sampler while the next one is the definition of the output files `withblank_output.dat`. As previously the output file is created and the two main functions are called with the same consequences: thanks to `setFieldSeparatorCharacter` and `setSeparatorCharacter`, used with the already discussed argument, the output keys are also usable with blank.

```
// Define the output
TString sOutFileName="withblank_output.dat";
TOutputFileKey *fout = new TOutputFileKey(sOutFileName);
// Set the separator to '=' so that the line could contains blank spaces (specific)
fout->setFieldSeparatorCharacter("=");
fout->setSeparatorCharacter("\n");
```

The rest of this macro is very usual, as one defines the code, the launcher and one runs it, one can check the content through a scan, shown below.xs

XIV.4.34.3 Console

```
Processing launchCodeFilesWithBlank.C...
************************************************************
*    Row   *  pressure * outpressu *   density * outdensit *
************************************************************
*        0 * 3.3622174 * 6.7244349 * 19.122886 * 57.368659 *
*        1 * 5.4302620 * 10.860524 * 0.0823283 * 0.2469849 *
*        2 * 6.2876717 * 12.575343 * 0.2559847 * 0.7679543 *
*        3 * 3.3986722 * 6.7973444 * 0.0503713 * 0.1511141 *
*        4 * 1.1833060 * 2.3666120 * 0.0364635 * 0.1093906 *
*        5 * 9.3162251 * 18.632450 * 0.0134744 * 0.0404234 *
*        6 * 2.1579343 * 4.3158686 * 0.0431745 * 0.1295236 *
*        7 * 1.2310214 * 2.4620429 * 12.797791 * 38.393373 *
*        8 * 4.5137647 * 9.0275294 * 2.3940563 * 7.1821691 *
*        9 * 8.4025508 * 16.805101 * 65.518819 * 196.55645 *
************************************************************
```

XIV.5 Macros Sensitivity

XIV.5.1 Macro "sensitivityBrutForceMethodFlowrate.C"

XIV.5.1.1 Objective

The objective of this macro is to perform a sensitivity analysis with brute force method on a set of eight parameters used in the *flowrate* model described in Section IV.1.2.1. Sensitivity indexes are computed dividing conditional variance by the standard deviation of the output variable.

Warning This macro is purely illustrative. It is not meant to be used for proper results with a real code / function as it needs a large number of computation to only get the first order index. Its main appeal is to be nicely illustrative: it shows plainly the definition of the conditional expectation and also its variance used to defined the first order sobol indices.

XIV.5.1.2 Macro Uranie

```
void drawBarWithTuple(TTree *tt, TString sx, TString sy, TString stitle)
{

  TLeaf *lx = tt->GetLeaf(sx);
  TLeaf *ly = tt->GetLeaf(sy);

  TH1F *hDiv = new TH1F("hDivdrawBarWithTuple",stitle,3,0,3);
  hDiv->SetCanExtend(TH1::kXaxis); //SetBit(TH1::kCanRebin);
  hDiv->SetStats(0);

  if ( hDiv ) {
    hDiv->SetBarWidth(0.45);
    hDiv->SetBarOffset(0.1);
    hDiv->SetMarkerColor(2);
    hDiv->SetMarkerSize(2);
    hDiv->SetFillColor(49);
    hDiv->SetTitle(stitle);
    for (Int_t i = 0; i < tt->GetEntries(); i++) {
      tt->GetEntry(i);
      TString title = *((string*)lx->GetValuePointer());
      hDiv->Fill(title, ly->GetValue());
   }
    hDiv->LabelsDeflate();
    hDiv->LabelsOption(">u");
    hDiv->SetMinimum(0.0);
    hDiv->SetMaximum(1.0);
    gStyle->SetPaintTextFormat("5.2f");
    hDiv->Draw("bar2, text45");
  }
}

void sensitivityBrutForceMethodFlowrate(Int_t nCond = 50, Int_t nbins = 10)
{

  // Create a TDataServer
  TDataServer * tds = new TDataServer();

  cout << endl << " *****************************************************" << endl;
  cout << " ** sensitivityBrutForceMethodFlowrate nbins[" << nbins << "] nCond[" << nCond ↩
      << "]" << endl;
  cout << " **" << endl;

  // Add the eight attributes of the study with uniform law
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

```
Int_t nvar = tds->getNAttributes();
cout << " ** nX[" << nvar << "]" << endl;

Int_t nS = nbins*nvar*nCond;
cout << " ** nS[" << nS << "]" << endl;

// TSampling *sam = new TSampling(tds, "lhs", nS);
TQMC * sam = new TQMC(tds, "halton", nS);
sam->generateSample();

// Load the function
gROOT->LoadMacro("UserFunctions.C");

// Create a TLauncherFunction from a TDataServer and an analytical function
// Rename the outpout attribute "ymod"
TLauncherFunction * tlf = new TLauncherFunction(tds, "flowrateModel","","ymod");
// Evaluate the function on all the design of experiments
tlf->setDrawProgressBar(kFALSE);
tlf->run();

TCanvas *Canvas = new TCanvas("c1", "Graph for the Macro modeler",5,64,1270,667);
TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
pad->Divide(2,2);
pad->cd(1);
tds->computeStatistic("ymod");
tds->draw("ymod");
Double_t dstdy = tds->getAttribute("ymod")->getStd();
Double_t svary = dstdy * dstdy;
cout << " ** ymod : std[" << dstdy << "] vary[" << svary  << "]" << endl;

tds->getAttribute("ymod")->setOutput();

gStyle->SetOptStat(1);

// Tempory TTree for Histogram visualisation
Double_t valSobolCrt;
string   sName;
TTree *tt = new TTree("sobolforcebrut","MonteCarlo brute force sobol sensitivity ");
tt->Branch("Var","string",&sName);
tt->Branch("Value",&valSobolCrt,"Value/D");
tt->SetMarkerColor(kRed);
tt->SetMarkerStyle(7);
tt->SetMarkerSize(1.75);

TCanvas *c = new TCanvas();
c->Divide(2);
c->cd(1);
for(Int_t ivar=0; ivar<nvar; ivar++) {
  cout << " *****************************" << endl << " *** " <<  tds->getAttribute(ivar) ↩
      ->GetName() << endl;

  const char * svar = tds->getAttribute(ivar)->GetName();

  if(ivar==0)
   pad->cd(2);
  else
   c->cd(1);

  tds->drawProfile(Form("ymod:%s", svar),"",Form("nclass=%d", nbins));
  TProfile *hprofs = (TProfile*)gPad->GetPrimitive(Form("Profile ymod:%s (Bin = %d )",  ↩
      svar, nbins+2));
```

```
    TNtupleD * ntd = new TNtupleD("dd", "sjsjs", "i:x:m");
    ntd->SetMarkerColor(kBlue);
    ntd->SetMarkerStyle(8);
    //    ntd->SetMarkerSize(1.25);
    Int_t nnbins = hprofs->GetNbinsX();
    for(Int_t i=1; i <= nnbins; i++)
      ntd->Fill(i-1, hprofs->GetBinCenter(i), hprofs->GetBinContent(i));

     tds->draw(Form("ymod:%s", svar));
     ntd->Draw("m:x", "","same");

    if(ivar==0)
      pad->cd(3);
    else
      c->cd(2);

    ntd->Draw("m");
    TH1F *htemp = (TH1F*)gPad->GetPrimitive("htemp");

    Double_t dvarcond = htemp->GetRMS();

    // Tempory TTree for histogram
    sName=string(svar);
    valSobolCrt = dvarcond*dvarcond /svary;
    tt->Fill();
    cout << " *** S1[ " << svar <<"] Cond. Var.[" << dvarcond*dvarcond << "] -- [" << ↩
        valSobolCrt <<"]" << endl;

    c->Modified(); c->Update(); c->SaveAs(Form("SAFlowRateVersus%s.png", svar));

    delete ntd;
  }
  pad->cd(4);
  drawBarWithTuple(tt, "Var", "Value", "Sensitivity Indexes : ymod  [ Brute-Force Method ]" ↩
      );


}
```

Each parameter is related to the `TDataServer` as a `TAttribute` and obeys an uniform law on specific interval:

```
TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

A design-of-experiments is built with a "Halton" method (n_S = 4000):

```
TQMC * sam = new TQMC(tds, "halton", nS);
sam->generateSample();
```

The function `flowrateModel` is loaded from the macro `UserFunctions.C` (the file can be found in `${URANIESYS}/share`

```
gROOT->LoadMacro("UserFunctions.C");
```

The `flowrateModel` model is applied on previous variables:

```
TLauncherFunction * tlf = new TLauncherFunction(tds, "flowrateModel","","ymod");
tlf->run();
```

Characteristic values for the output attribute are computed:

```
tds->computeStatistic("ymod");
```

Sensitivity indexes are computed in the for loop. Average value of output variable is computed on nbins+2=12 points for each input variable:

```
c->cd(1);
...
Int_t nnbins = hprofs->GetNbinsX();
for(Int_t i=1; i <= nnbins; i++)
ntd->Fill(i-1, hprofs->GetBinCenter(i), hprofs->GetBinContent(i));
```

The RMS value is obtained from the graphic of ymod versus the considered output variable and the sensitivity index is computed dividing the conditional variance value by the standard deviation of the output variable **ymod**.

```
c->cd(2);
ntd->Draw("m");
TH1F *htemp = (TH1F*)gPad->GetPrimitive("htemp");
Double_t dvarcond = htemp->GetRMS();
valSobolCrt = dvarcond*dvarcond /svary;
```

XIV.5.1.3 Graph

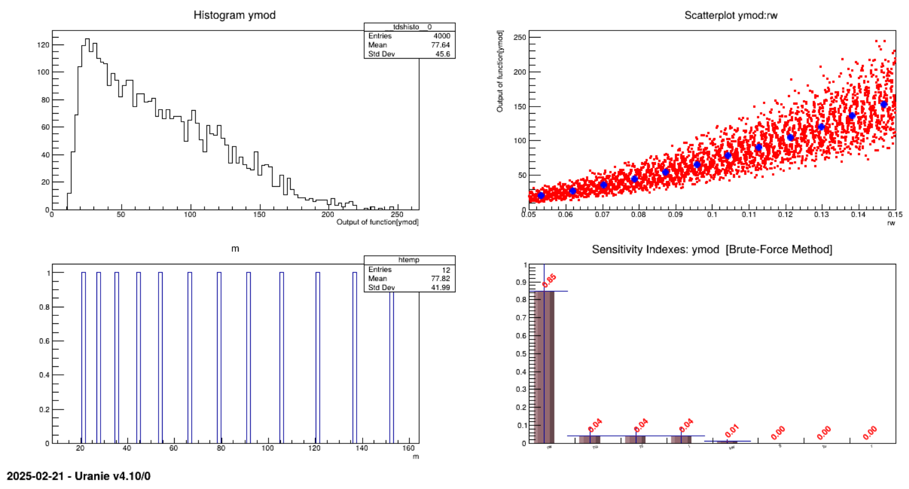


Figure XIV.44: Graph of the macro "sensitivityBrutForceMethodFlowrate.C"

XIV.5.1.4 Console

```
Processing sensitivityBrutForceMethodFlowrate.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025
```

```
 ******************************************************
 ** sensitivityBrutForceMethodFlowrate nbins[10] nCond[50]
 **
 ** nX[8]
 ** nS[4000]
 ** ymod : std[45.6059] vary[2079.9]
 *****************************
 *** rw
 *** S1[ rw] Cond. Var.[1763.28] -- [0.847774]
Info in <TCanvas::Print>: png file SAFlowRateVersusrw.png has been created
 *****************************
 *** r
 *** S1[ r] Cond. Var.[0.0624988] -- [3.00489e-05]
Info in <TCanvas::Print>: png file SAFlowRateVersusr.png has been created
 *****************************
 *** tu
 *** S1[ tu] Cond. Var.[0.0886501] -- [4.26223e-05]
Info in <TCanvas::Print>: png file SAFlowRateVersustu.png has been created
 *****************************
 *** tl
 *** S1[ tl] Cond. Var.[0.0909604] -- [4.37331e-05]
Info in <TCanvas::Print>: png file SAFlowRateVersustl.png has been created
 *****************************
 *** hu
 *** S1[ hu] Cond. Var.[88.368] -- [0.0424867]
Info in <TCanvas::Print>: png file SAFlowRateVersushu.png has been created
 *****************************
 *** hl
 *** S1[ hl] Cond. Var.[88.2039] -- [0.0424078]
Info in <TCanvas::Print>: png file SAFlowRateVersushl.png has been created
 *****************************
 *** l
 *** S1[ l] Cond. Var.[84.9543] -- [0.0408454]
Info in <TCanvas::Print>: png file SAFlowRateVersusl.png has been created
 *****************************
 *** kw
 *** S1[ kw] Cond. Var.[20.8848] -- [0.0100413]
Info in <TCanvas::Print>: png file SAFlowRateVersuskw.png has been created
```

XIV.5.2 Macro "sensitivityFiniteDifferencesFunctionFlowrate.C"

XIV.5.2.1 Objective

The objective of this macro is to compute the finite differences indexes on a function.

XIV.5.2.2 Macro Uranie

```
{
  // loading the flowrateModel function
  gROOT->LoadMacro("UserFunctions.C");

  // Define the DataServer  and add the attributes (stochastic variables here)
  TDataServer *tds = new TDataServer("tdsflowrate", "DataBase flowrate");
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));


  tds->getAttribute("rw")->setDefaultValue(0.075);
  tds->getAttribute("r")->setDefaultValue(25000.0);
  tds->getAttribute("tu")->setDefaultValue(90000.0);
  tds->getAttribute("tl")->setDefaultValue(90.0);
  tds->getAttribute("hu")->setDefaultValue(1050.0);
  tds->getAttribute("hl")->setDefaultValue(760.0);
  tds->getAttribute("l")->setDefaultValue(1400.0);
  tds->getAttribute("kw")->setDefaultValue(10500.0);

 // Create a TFiniteDifferences object
 TFiniteDifferences * tfindef = new TFiniteDifferences(tds,"flowrateModel", "rw:r:tu:tl:hu: ↩
    hl:l:kw", "y", "steps=1%");
 tfindef->setDrawProgressBar(kFALSE);
 tfindef->computeIndexes();
 TMatrixD matRes = tfindef->getSensitivityMatrix();
 matRes.Print();


}
```

The function `flowrateModel` is loaded from the macro `UserFunctions.C` (the file can be found in `${URANIESYS}/`

```
gROOT->LoadMacro("UserFunctions.C");
```

Each parameter is related to the `TDataServer` as a `TAttribute` and obeys an uniform law on specific interval:

```
TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

Each parameter gets a default value:

```
tds->getAttribute("rw")->setDefaultValue(0.075);
tds->getAttribute("r")->setDefaultValue(25000.0);
tds->getAttribute("tu")->setDefaultValue(90000.0);
tds->getAttribute("tl")->setDefaultValue(90.0);
tds->getAttribute("hu")->setDefaultValue(1050.0);
tds->getAttribute("hl")->setDefaultValue(760.0);
tds->getAttribute("l")->setDefaultValue(1400.0);
tds->getAttribute("kw")->setDefaultValue(10500.0);
```

To instantiate the `TFiniteDifferences` object, one uses the `TDataServer`, the name of the function, the name of the output of the function, the names of the input variables separated by ":" and the option to specify the sampling:

```
TFiniteDifferences * tfindef = new TFiniteDifferences(tds, "flowrateModel", "rw:r:tu:tl:hu: ↩
    hl:l:kw","y", "steps=1%");
```

Computation of sensitivity indexes:

```
tfindef->computeIndexes();
```

XIV.5.2.3 Console

```
Processing sensitivityFiniteDifferencesFunctionFlowrate.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025


1x8 matrix is as follows

     |      0    |      1    |      2    |      3    |      4    |
----------------------------------------------------------------------
   0 |       1019  -3.586e-07   1.265e-09    0.001265      0.1321


     |      5    |      6    |      7    |
----------------------------------------------------------------------
   0 |    -0.1321    -0.02729    0.003639
```

XIV.5.3 Macro "**sensitivityDataBaseFlowrate.C**"

XIV.5.3.1 Objective

The objective of this macro is to perform a SRC regression on data stored in a `TDataServer`. Data are loaded in the `TDataServer` from an ASCII data file `flowrateUniformDesign.dat`:

```
#NAME: flowrateborehole
#TITLE: Uniform design of flow rate borehole problem proposed by Ho and Xu(2000)
#COLUMN_NAMES: rw| r| tu| tl| hu| hl| l| kw | ystar
#COLUMN_TITLES: r_{#omega}| r | T_{u} | T_{l} | H_{u} | H_{l} | L | K_{#omega} | y^{*}
#COLUMN_UNITS: m | m | m^{2}/yr | m^{2}/yr | m | m | m | m/yr | m^{3}/yr

0.0500 33366.67  63070.0 116.00 1110.00 768.57 1200.0 11732.14  26.18
0.0500   100.00  80580.0  80.73 1092.86 802.86 1600.0 10167.86  14.46
0.0567   100.00  98090.0  80.73 1058.57 717.14 1680.0 11106.43  22.75
0.0567 33366.67  98090.0  98.37 1110.00 734.29 1280.0 10480.71  30.98
0.0633   100.00 115600.0  80.73 1075.71 751.43 1600.0 11106.43  28.33
0.0633 16733.33  80580.0  80.73 1058.57 785.71 1680.0 12045.00  24.60
0.0700 33366.67  63070.0  98.37 1092.86 768.57 1200.0 11732.14  48.65
0.0700 16733.33 115600.0 116.00  990.00 700.00 1360.0 10793.57  35.36
0.0767   100.0  115600.0  80.73 1075.71 751.43 1520.0 10793.57  42.44
0.0767 16733.33  80580.0  80.73 1075.71 802.86 1120.0  9855.00  44.16
0.0833 50000.00  98090.0  63.10 1041.43 717.14 1600.0 10793.57  47.49
0.0833 50000.00 115600.0  63.10 1007.14 768.57 1440.0 11419.29  41.04
0.0900 16733.33  63070.0 116.00 1075.71 751.43 1120.0 11419.29  83.77
0.0900 33366.67 115600.0 116.00 1007.14 717.14 1360.0 11106.43  60.05
0.0967 50000.00  80580.0  63.10 1024.29 820.00 1360.0  9855.00  43.15
0.0967 16733.33  80580.0  98.37 1058.57 700.00 1120.0 10480.71  97.98
0.1033 50000.00  80580.0  63.10 1024.29 700.00 1520.0 10480.71  74.44
0.1033 16733.33  80580.0  98.37 1058.57 820.00 1120.0 10167.86  72.23
0.1100 50000.00  98090.0  63.10 1024.29 717.14 1520.0 10793.57  82.18
```

```
0.1100   100.00  63070.0  98.37 1041.43 802.86 1600.0 12045.00  68.06
0.1167 33366.67  63070.0 116.00  990.00 785.71 1280.0 12045.00  81.63
0.1167   100.00  98090.0  98.37 1092.86 802.86 1680.0  9855.00  72.5
0.1233 16733.33 115600.0  80.73 1092.86 734.29 1200.0 11419.29 161.35
0.1233 16733.33  63070.0  63.10 1041.43 785.71 1680.0 12045.00  86.73
0.1300 33366.67  80580.0 116.00 1110.00 768.57 1280.0 11732.14 164.78
0.1300   100.00  98090.0  98.37 1110.00 820.00 1280.0 10167.86 121.76
0.1367 50000.00  98090.0  63.10 1007.14 820.00 1440.0 10167.86  76.51
0.1367 33366.67  98090.0 116.00 1024.29 700.00 1200.0 10480.71 164.75
0.1433 50000.00  63070.0 116.00  990.00 785.71 1440.0  9855.00  89.54
0.1433 50000.00 115600.0  63.10 1007.14 734.29 1440.0 11732.14 141.09
0.1500 33366.67  63070.0  98.37  990.00 751.43 1360.0 11419.29 139.94
0.1500   100.00 115600.0  80.73 1041.43 734.29 1520.0 11106.43 157.59
```

XIV.5.3.2 Macro Uranie

```
{
  // Create a TDataServer
  TDataServer * tds = new TDataServer();
  // Load a database in an ASCII file
  tds->fileDataRead("flowrateUniformDesign.dat");

  // Graph
  TCanvas  *Canvas = new TCanvas("c2", "Graph for the Macro",5,64,1270,667);
  // Visualisation
  tds->Draw("ystar:rw");

  // Sensitivity analysis
  TRegression *treg = new TRegression(tds, "rw:r:tu:tl:hu:hl:l:kw", "ystar", "src");
  treg->computeIndexes();

  treg->drawIndexes("Flowrate", "", "hist,first");
  //treg->getResultTuple()->Scan();

  // Graph
  TCanvas *c = (TCanvas *)(gROOT->FindObject("__sensitivitycan__0"));
  TCanvas  *can = new TCanvas("c1", "Graph for the Macro sensitivityDataBaseFlowrate" ↩
      ,5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2);
  pad->cd(1);
  Canvas->DrawClonePad();
  pad->cd(2);
  c->DrawClonePad();

}
```

The `TDataServer` is filled with the data file `flowrateUniformDesign.dat` through the `fileDataRead` method:

```
tds->fileDataRead("flowrateUniformDesign.dat");
```

The regression is performed on all the variables with a SRC method and sensitivity indexes are computed:

```
TRegression *treg = new TRegression(tds, "rw:r:tu:tl:hu:hl:l:kw", "ystar", "src");
treg->computeIndexes();
```

XIV.5.3.3 Graph

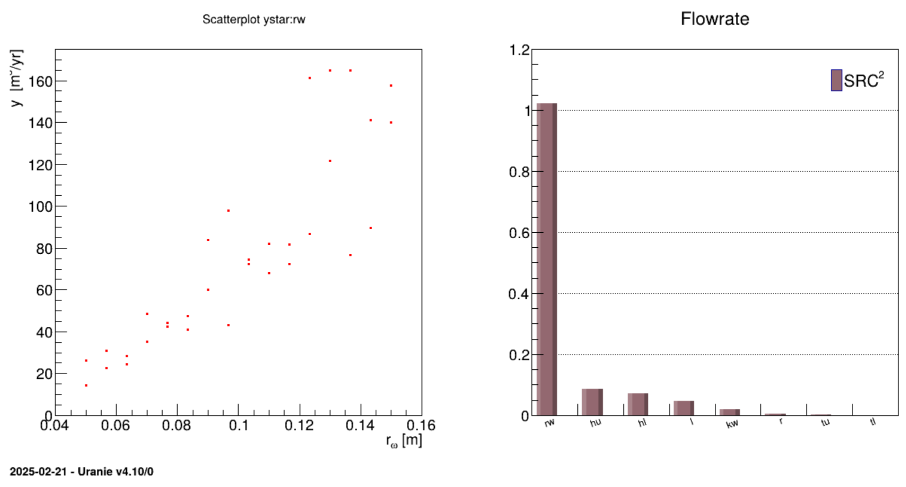


Figure XIV.45: Graph of the macro "sensitivityDataBaseFlowrate.C"

XIV.5.4 Macro "sensitivityFASTFunctionFlowrate.C"

XIV.5.4.1 Objective

The objective of this macro is to perform a Fast sensitivity analysis on a set of eight parameters used in the flowrateModel model described in Section IV.1.2.1.

XIV.5.4.2 Macro Uranie

```
void sensitivityFASTFunctionFlowrate(){

  gROOT->LoadMacro("UserFunctions.C");

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  // \param Size of a sampling.
  Int_t nS = 4000;
  // Graph
  TFast * tfast = new TFast(tds, "flowrateModel", nS);
  tfast->setDrawProgressBar(kFALSE);
  tfast->computeIndexes("graph");

  tfast->getResultTuple()->Scan("Out:Inp:Order:Method:Value","Algo==\"--first--\"");

}
```

The function `flowrateModel` is loaded from the macro `UserFunctions.C` (the file can be found in `${URANIESYS}/`

```
gROOT->LoadMacro("UserFunctions.C");
```

Each parameter is related to the `TDataServer` as a `TAttribute` and obeys an uniform law on specific interval:

```
TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

To instantiate the `TFast` object, one uses the `TDataServer`, the name of the function and the number of samplings needed to perform sensitivity analysis (here nS=500):

```
TFast * tfast = new TFast(tds, "flowrateModel", nS);
```

Computation of sensitivity indexes:

```
tfast->computeIndexes();
```

XIV.5.4.3 Graph

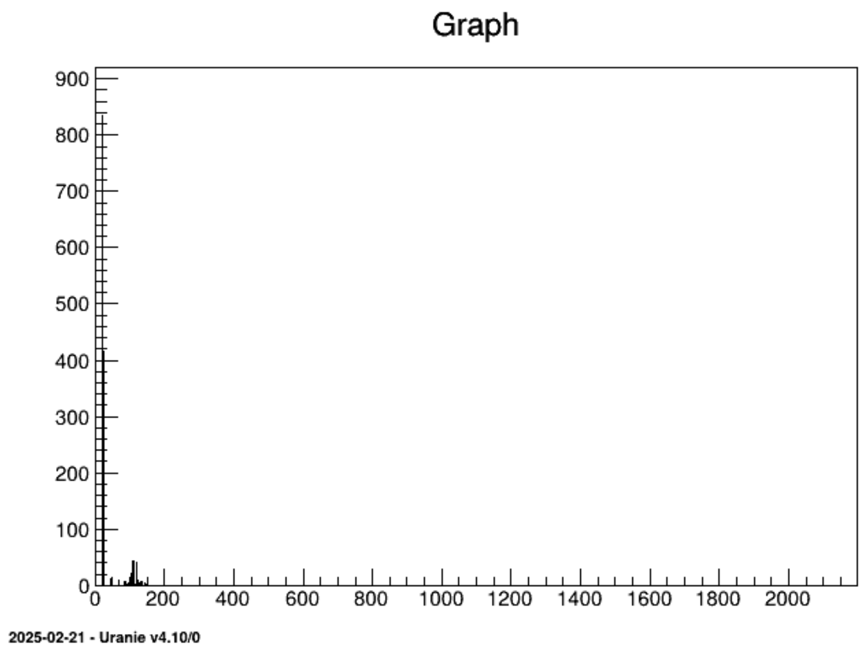


Figure XIV.46: Graph of the macro "`sensitivityFASTFunctionFlowrate.C`"

XIV.5.4.4 Console

```
Processing sensitivityFASTFunctionFlowrate.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8530]
 <URANIE::WARNING> TDataServer::getTuple Error : There is no tree!
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8530]
 <URANIE::WARNING> TDataServer::getTuple Error : There is no tree!
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${SOURCEDIR}/meTIER/sampler/souRCE/TSpaceFilling.cxx] Line[167]
 <URANIE::INFO> TSamplerStochastic::init: the TDS [tdsflowreate] contains data: we need to ↩
    empty it !
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>
************************************************************************
*    Row   *        Out *        Inp *      Order *     Method *      Value *
************************************************************************
*        0 * flowrateM *         rw *      First *       FAST * 0.8278187 *
*        2 * flowrateM *          r *      First *       FAST * 8.924e-07 *
*        4 * flowrateM *         tu *      First *       FAST * 2.308e-06 *
*        6 * flowrateM *         tl *      First *       FAST * 3.204e-05 *
*        8 * flowrateM *         hu *      First *       FAST * 0.0414390 *
*       10 * flowrateM *         hl *      First *       FAST * 0.0414046 *
*       12 * flowrateM *          l *      First *       FAST * 0.0392873 *
*       14 * flowrateM *         kw *      First *       FAST * 0.0094983 *
************************************************************************
==> 8 selected entries
```

XIV.5.5 Macro "sensitivityRBDFunctionFlowrate.C"

XIV.5.5.1 Objective

The objective of this macro is to perform a RBD sensitivity analysis on a set of eight parameters used in the `flowrateModel` model described in Section IV.1.2.1.

XIV.5.5.2 Macro Uranie

```
void sensitivityRBDFunctionFlowrate(){

  gROOT->LoadMacro("UserFunctions.C");

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsflowrate", "DataBase flowrate");
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  // \param Size of a sampling.
  Int_t nS = 4000;
  // Graph
  TRBD * trbd = new TRBD(tds, "flowrateModel", nS);
  trbd->setDrawProgressBar(kFALSE);
  trbd->computeIndexes("graph");

  trbd->getResultTuple()->Scan("Out:Inp:Order:Method:Value","Algo==\"--first--\"");

}
```

The function `flowrateModel` is loaded from the macro `UserFunctions.C` (the file can be found in `${URANIESYS}/`

```
gROOT->LoadMacro("UserFunctions.C");
```

Each parameter is related to the `TDataServer` as a `TAttribute` and obeys an uniform law on specific interval

```
TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

To instantiate the `TRBD` object, one uses the `TDataServer`, the name of the function and the number of samplings needed to perform sensitivity analysis (here n_S=4000):

```
TRBD * trbd = new TRBD(tds, "flowrateModel", nS);
```

Computation of sensitivity indexes:

```
trbd->computeIndexes();
```

XIV.5.5.3 Graph

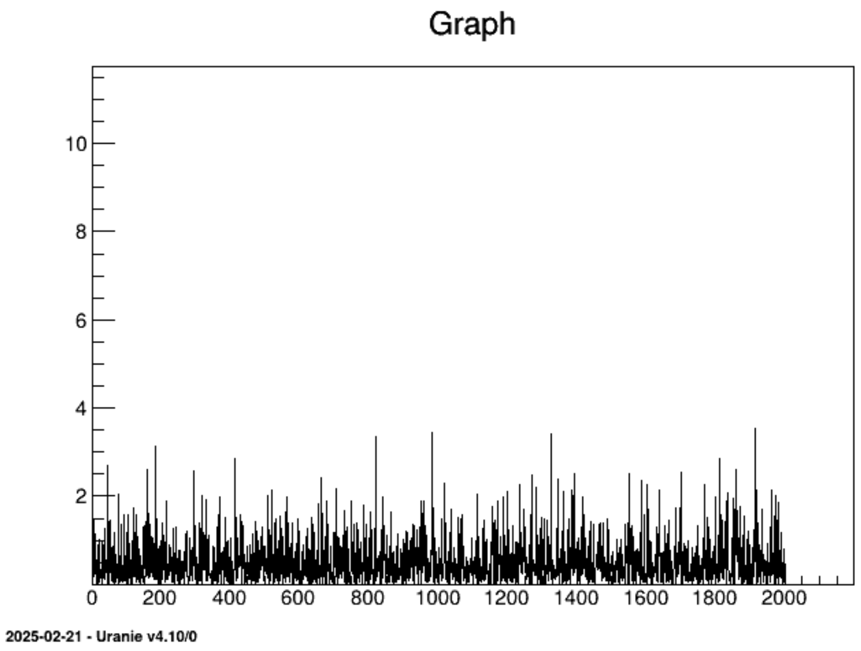


Figure XIV.47: Graph of the macro "sensitivityRBDFunctionFlowrate.C"

XIV.5.5.4 Console

```
Processing sensitivityRBDFunctionFlowrate.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8530]
 <URANIE::WARNING> TDataServer::getTuple Error : There is no tree!
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8530]
 <URANIE::WARNING> TDataServer::getTuple Error : There is no tree!
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${SOURCEDIR}/meTIER/sampler/souRCE/TSpaceFilling.cxx] Line[167]
 <URANIE::INFO> TSamplerStochastic::init: the TDS [tdsflowrate] contains data: we need to  ↩
    empty it !
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>
************************************************************************
*    Row   *       Out *       Inp *     Order *    Method *     Value *
************************************************************************
```

```
*        0 * flowrateM *        rw *     First *       RBD * 0.7558010 *
*        2 * flowrateM *         r *     First *       RBD * 0.0026080 *
*        4 * flowrateM *        tu *     First *       RBD * 0.0035324 *
*        6 * flowrateM *        tl *     First *       RBD * 0.0032848 *
*        8 * flowrateM *        hu *     First *       RBD * 0.0408758 *
*       10 * flowrateM *        hl *     First *       RBD * 0.0469345 *
*       12 * flowrateM *         l *     First *       RBD * 0.0347870 *
*       14 * flowrateM *        kw *     First *       RBD * 0.0165843 *
************************************************************************
==> 8 selected entries
```

XIV.5.6 Macro "sensitivityMorrisFunctionFlowrate.C"

XIV.5.6.1 Objective

The objective of this macro is to perform a Morris sensitivity analysis on a set of eight parameters used in the `flowrateModel` model described in Section IV.1.2.1.

XIV.5.6.2 Macro Uranie

```
void sensitivityMorrisFunctionFlowrate(Int_t nk = 5)
{

  gROOT->LoadMacro("UserFunctions.C");

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  Int_t nreplique = 3;
  Int_t nlevel = 10;
  TMorris * scmo = new TMorris(tds, "flowrateModel", nreplique, nlevel);
  scmo->setDrawProgressBar(kFALSE);
  scmo->generateSample();

  tds->exportData("_morris_sampling_.dat");
  scmo->computeIndexes();

  tds->exportData("_morris_launching_.dat");

  TTree *ntresu = scmo->getMorrisResults();
  ntresu->Scan("*");

  // Graph
  TCanvas  *cc = new TCanvas("c1", "Graph for the Macro sensitivityMorrisFunctionFlowrate" ↩
      ,5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2);
  pad->cd(1);
  scmo->drawSample("", -1,"nonewcanv");
  pad->cd(2);
  scmo->drawIndexes("mustar,nonewcanv");

}
```

The function `flowrateModel` is loaded from the macro `UserFunctions.C` (the file can be found in `${URANIESYS}/share`

```
gROOT->LoadMacro("UserFunctions.C");
```

Each parameter is related to the `TDataServer` as a `TAttribute` and obeys an uniform law on specific interval:

```
TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

To instantiate the `TMorris` , one uses the `TDataServer`, the name of the function, the number of replicas (here nreplique=3), the level parameter (here nlevel=10)

```
TMorris * scmo = new TMorris(tds, "flowrateModel", nreplique, nlevel);
```

Creation of the sampling:

```
scmo->generateSample();
```

Data are exported in an ASCII file:

```
tds->exportData("_morris_sampling_.dat");
```

Computation of sensitivity indexes:

```
scmo->computeIndexes();
```

XIV.5.6.3 Graph

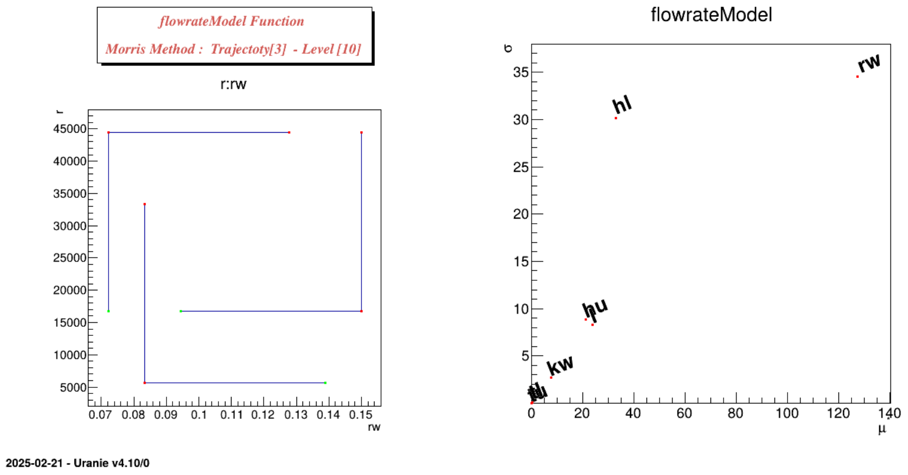


Figure XIV.48: Graph of the macro "`sensitivityMorrisFunctionFlowrate.C`"

XIV.5.6.4 Console

```
Processing sensitivityMorrisFunctionFlowrate.C...
************************************************************************
*    Row   *      Input *     Output *      mu.mu * mustar.mu * sigma.sig *
************************************************************************
*        0 *         rw * flowrateM * 127.47900 * 127.47900 * 34.521839 *
*        1 *          r * flowrateM * -0.069601 * 0.0696013 * 0.0793689 *
*        2 *         tu * flowrateM * 0.0004201 * 0.0004201 * 0.0004641 *
*        3 *         tl * flowrateM * 0.4659763 * 0.4659763 * 0.3301256 *
*        4 *         hu * flowrateM * 21.192361 * 21.192361 * 8.8498989 *
*        5 *         hl * flowrateM * -32.74887 * 32.748874 * 30.146134 *
*        6 *          l * flowrateM * -23.89328 * 23.893280 * 8.2781934 *
*        7 *         kw * flowrateM * 7.5766167 * 7.5766167 * 2.7457665 *
************************************************************************
```

XIV.5.7 Macro "sensitivityMorrisFunctionFlowrateRunner.C"

XIV.5.7.1 Objective

The objective of this macro is to perform a Morris sensitivity analysis on a set of eight parameters used in the flowrateModel model described in Section IV.1.2.1, but this time using the Relauncher architecture.

XIV.5.7.2 Macro Uranie

```
void sensitivityMorrisFunctionFlowrateRunner(Int_t nk = 5)
{

  gROOT->LoadMacro("UserFunctions.C");

  // Define the attributes
  TUniformDistribution rw("rw", 0.05, 0.15);
  TUniformDistribution r("r", 100.0, 50000.0);
  TUniformDistribution tu("tu", 63070.0, 115600.0);
  TUniformDistribution tl("tl", 63.1, 116.0);
  TUniformDistribution hu("hu", 990.0, 1110.0);
  TUniformDistribution hl("hl", 700.0, 820.0);
  TUniformDistribution l("l", 1120.0, 1680.0);
  TUniformDistribution kw("kw", 9855.0, 12045.0);

  // Create the evaluator
  TCIntEval code("flowrateModel");
  // Create output attribute
  TAttribute yout("flowrateModel");
  // Provide input/output attributes to the assessor
  code.setInputs(8, &rw, &r, &tu, &tl, &hu, &hl, &l, &kw);
  code.setOutputs(1, &yout);

  TSequentialRun run(&code); // To be replaced to distribute the computation
  run.startSlave();
  if( run.onMaster())
  {
    // Create the dataserver
    TDataServer *tds = new TDataServer("sobol", "foo bar pouet chocolat");
    tds->addAttribute(&rw);
    tds->addAttribute(&r);
    tds->addAttribute(&tu);
    tds->addAttribute(&tl);
    tds->addAttribute(&hu);
    tds->addAttribute(&hl);
    tds->addAttribute(&l);
    tds->addAttribute(&kw);
    Int_t nreplique = 3;
    Int_t nlevel = 10;
    // Create the Morris object
    TMorris * scmo = new TMorris(tds, &run, nreplique, nlevel);
    scmo->setDrawProgressBar(kFALSE);
    scmo->generateSample();

    tds->exportData("_morris_sampling_.dat");
    scmo->computeIndexes();

    tds->exportData("_morris_launching_.dat");

    TTree *ntresu = scmo->getMorrisResults();
    ntresu->Scan("*");

    // Graph
    TCanvas  *cc = new TCanvas("c1", "Graph for the Macro  ↩
        sensitivityMorrisFunctionFlowrateRunner",5,64,1270,667);
    TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
    pad->Divide(2);
    pad->cd(1);
    scmo->drawSample("", -1,"nonewcanv");
    pad->cd(2);
    scmo->drawIndexes("mustar,nonewcanv");

  }
}
```

The function `flowrateModel` is loaded from the macro `UserFunctions.C` (the file can be found in `${URANIESYS}/share`

```
gROOT->LoadMacro("UserFunctions.C");
```

Each parameter is related to the `TDataServer` as a `TAttribute` and obeys an uniform law on specific interval:

```
// Define the attributes
TUniformDistribution rw("rw", 0.05, 0.15);
TUniformDistribution r("r", 100.0, 50000.0);
TUniformDistribution tu("tu", 63070.0, 115600.0);
TUniformDistribution tl("tl", 63.1, 116.0);
TUniformDistribution hu("hu", 990.0, 1110.0);
TUniformDistribution hl("hl", 700.0, 820.0);
TUniformDistribution l("l", 1120.0, 1680.0);
TUniformDistribution kw("kw", 9855.0, 12045.0);
```

The interface to the function is then defined, using the Relauncher interface, through a `TCIntEval` object and a sequential runner:

```
 // Create the evaluator
TCIntEval code("flowrateModel");
// Create output attribute
TAttribute yout("flowrateModel");
// Provide input/output attributes to the assessor
code.setInputs(8, &rw, &r, &tu, &tl, &hu, &hl, &l, &kw);
code.setOutputs(1, &yout);

TSequentialRun run(&code); // To be replaced to distribute the computation
run.startSlave();
```

The dataserver object is defined only on the master to avoid useless replication if one wants to run the estimation of the function in parallel (by changing the `TSequentialRun` by either a `TThreadedRun` or a `TMpiRun`). To instantiate the `TMorris` object, one uses the `TDataServer`, a pointer to the chosen runner, the number of replicas (here nreplique=3), the level parameter (here nlevel=10)

```
TMorris * scmo = new TMorris(tds, &run, nreplique, nlevel);
```

Creation of the sampling:

```
scmo->generateSample();
```

Data are exported in an ASCII file:

```
tds->exportData("_morris_sampling_.dat");
```

Computation of sensitivity indexes:

```
scmo->computeIndexes();
```

The rest of the code is providing command to get a final plot.

XIV.5.7.3 Graph

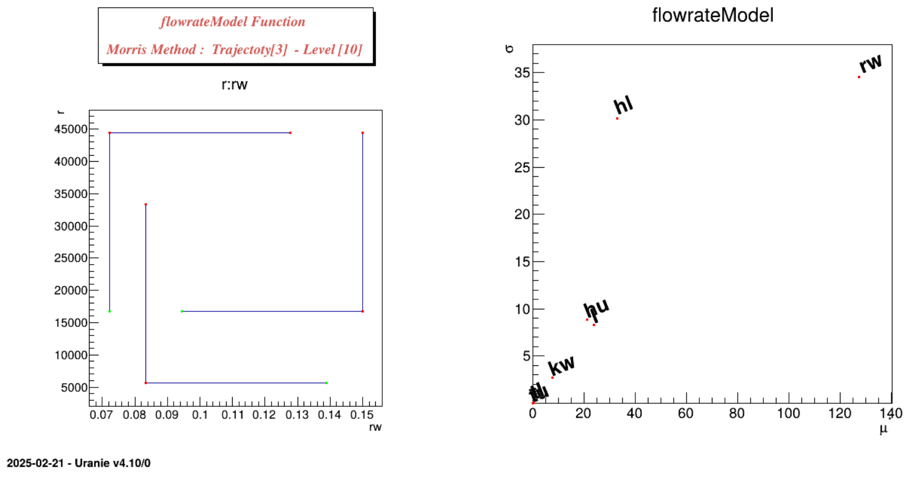


Figure XIV.49: Graph of the macro "sensitivityMorrisFunctionFlowrateRunner.C"

XIV.5.7.4 Console

```
Processing sensitivityMorrisFunctionFlowrateRunner.C...
**************************************************************************
*    Row   *      Input *    Output *      mu.mu * mustar.mu * sigma.sig *
**************************************************************************
*        0 *         rw * flowrateM * 127.47900 * 127.47900 * 34.521839 *
*        1 *          r * flowrateM * -0.069601 * 0.0696013 * 0.0793689 *
*        2 *         tu * flowrateM * 0.0004201 * 0.0004201 * 0.0004641 *
*        3 *         tl * flowrateM * 0.4659763 * 0.4659763 * 0.3301256 *
*        4 *         hu * flowrateM * 21.192361 * 21.192361 * 8.8498989 *
*        5 *         hl * flowrateM * -32.74887 * 32.748874 * 30.146134 *
*        6 *          l * flowrateM * -23.89328 * 23.893280 * 8.2781934 *
*        7 *         kw * flowrateM * 7.5766167 * 7.5766167 * 2.7457665 *
**************************************************************************
```

XIV.5.8 Macro "**sensitivityRegressionFunctionFlowrate.C**"

XIV.5.8.1 Objective

The objective of this macro is to perform a regression with "SRC" method on a database generated with a function using sampling of parameters obeying uniform laws with 4000 patterns. `flowrateModel` is a function defined in Section IV.1.2.1 and "loaded" through the macro `UserFunctions.C` (the file can be found in `${URANIESYS}/share/uranie/m`
Function `flowrateModel` uses the eight variables defined in Section IV.1.2.1 and set in the main macro.

XIV.5.8.2 Macro Uranie

```
void sensitivityRegressionFunctionFlowrate(){

  gROOT->LoadMacro("UserFunctions.C");

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  // \param Size of a sampling.
  Int_t nS = 4000;

  TSampling *sampling = new TSampling(tds, "lhs", nS);
  sampling->generateSample();


  TLauncherFunction * tlf = new TLauncherFunction(tds, "flowrateModel");
  tlf->setDrawProgressBar(kFALSE);
  tlf->run();


  TRegression * treg = new TRegression(tds, "rw:r:tu:tl:hu:hl:l:kw","flowrateModel", "SRC") ↩
      ;
  treg->computeIndexes();
  treg->getResultTuple()->SetScanField(60);

  treg->getResultTuple()->Scan("Out:Inp:Method:Algo:Value:CILower:CIUpper","Order==\"First ↩
      \"");


  TCanvas  *can = new TCanvas("c1", "Graph for the Macro  ↩
      sensitivityRegressionFunctionFlowrate",5,64,1270,667);
  treg->drawIndexes("Flowrate", "", "hist,first,nonewcanv");

}
```

Each attribute is related to a `TAttribute` obeying uniform laws on specific intervals:

```
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
```

```
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

The sampling is generated on 4000 patterns with a LHS method:

```
TSampling *sampling = new TSampling(tds, "lhs", 4000);
sampling->generateSample();
```

Function `flowrateModel` is set to perform calculation on the sampling:

```
TLauncherFunction * tlf = new TLauncherFunction(tds, "flowrateModel");
tlf->run();
```

The regression is performed over all variables:

```
TRegression * treg = new TRegression(tds, "rw:r:tu:tl:hu:hl:l:kw","flowrateModel", "SRRC");
treg->computeIndexes();
```

Sensitivity indexes are then displayed through an histogram and a pie graph:

```
TCanvas *cc = new TCanvas("canhist", "histgramme");
treg->drawIndexes("Flowrate", "", "nonewcanv,hist,first");
TCanvas *ccc = new TCanvas("canpie", "TPie");
treg->drawIndexes("Flowrate", "", "nonewcanv,pie,first");
```

XIV.5.8.3 Graph

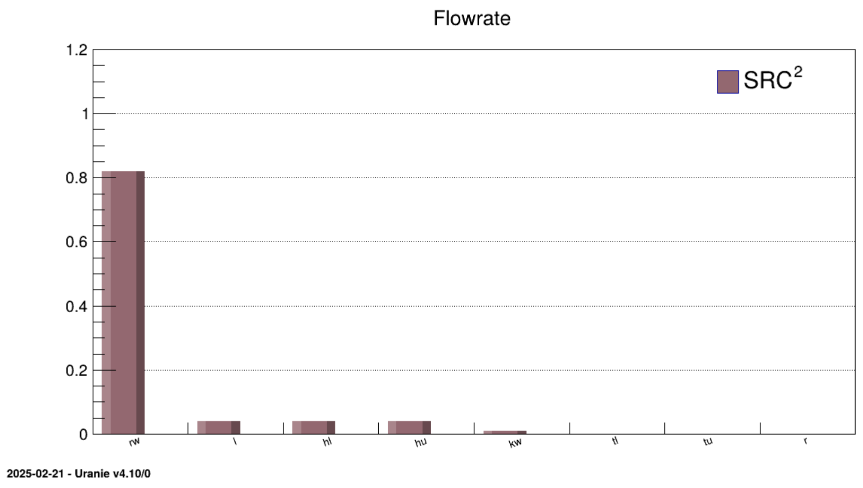


Figure XIV.50: Graph of the macro "`sensitivityRegressionFunctionFlowrate.C`"

XIV.5.8.4 Console

```
Processing sensitivityRegressionFunctionFlowrate.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                       Copyright (C) 2013-2025 CEA/DES
                       Contact: support-uranie@cea.fr
```

```
                      Date: Fri Feb 21, 2025

************************************************************************************
*    Row   *    Out *       Inp * Metho *      Algo *    Value *  CILower *  CIUpper *
************************************************************************************
*        0 * flowra *        rw * SRC^2 * --first-- * 0.820265 *        -1 *        -1 *
*        2 * flowra *        rw * SRC^2 * --rho^2-- *  0.81668 * 0.805846 * 0.826888 *
*        4 * flowra *         r * SRC^2 * --first-- * 5.97e-06 *        -1 *        -1 *
*        6 * flowra *         r * SRC^2 * --rho^2-- * 1.92e-06 * 2.65e-07 * 0.001339 *
*        8 * flowra *        tu * SRC^2 * --first-- * 8.64e-06 *        -1 *        -1 *
*       10 * flowra *        tu * SRC^2 * --rho^2-- * 4.03e-06 * 3.14e-07 * 0.001306 *
*       12 * flowra *        tl * SRC^2 * --first-- * 5.73e-05 *        -1 *        -1 *
*       14 * flowra *        tl * SRC^2 * --rho^2-- * 0.000209 * 7.34e-07 * 0.002085 *
*       16 * flowra *        hu * SRC^2 * --first-- * 0.039645 *        -1 *        -1 *
*       18 * flowra *        hu * SRC^2 * --rho^2-- * 0.037119 * 0.026221 * 0.049000 *
*       20 * flowra *        hl * SRC^2 * --first-- * 0.040597 *        -1 *        -1 *
*       22 * flowra *        hl * SRC^2 * --rho^2-- * 0.039708 * 0.028688 * 0.052470 *
*       24 * flowra *         l * SRC^2 * --first-- * 0.040895 *        -1 *        -1 *
*       26 * flowra *         l * SRC^2 * --rho^2-- * 0.041241 * 0.029896 * 0.054454 *
*       28 * flowra *        kw * SRC^2 * --first-- * 0.009174 *        -1 *        -1 *
*       30 * flowra *        kw * SRC^2 * --rho^2-- * 0.009090 * 0.004191 * 0.015767 *
*       32 * flowra *   __sum__ * SRC^2 * --first-- *  0.95065 *        -1 *        -1 *
*       34 * flowra *    __R2__ * SRC^2 * --first-- * 0.947296 *        -1 *        -1 *
*       36 * flowra *   __R2A__ * SRC^2 * --first-- *  0.94719 *        -1 *        -1 *
************************************************************************************
==> 19 selected entries
```

XIV.5.9 Macro "sensitivitySobolFunctionFlowrate.C"

XIV.5.9.1 Objective

The objective of this macro is to perform Sobol sensitivity analysis on a set of eight parameters used in the flowrateModel model described in Section IV.1.2.1.

XIV.5.9.2 Macro Uranie

```
{

  gROOT->LoadMacro("UserFunctions.C");

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  Int_t ns = 100000;
  TSobol * tsobol = new TSobol(tds, "flowrateModel", ns, "rw:r:tu:tl:hu:hl:l:kw", " ↩
      flowrateModel", "pouet");
  tsobol->setDrawProgressBar(kFALSE);
  tsobol->computeIndexes();
  tsobol->getResultTuple()->Scan("*","Algo==\"--first--\" || Algo==\"--total--\"");

  TCanvas *cc = new TCanvas("c1", "histgramme",5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2,1);
  pad->cd(1);
  tsobol->drawIndexes("Flowrate", "", "nonewcanv,hist,all");

  pad->cd(2);
  tsobol->drawIndexes("Flowrate", "", "nonewcanv,pie,first");

  gSystem->Rename("_sobol_launching_.dat","ref_sobol_launching_.dat");
  tds->exportData("_onlyMandN_sobol_launching_.dat","rw:r:tu:tl:hu:hl:l:kw:flowrateModel"," ↩
      sobol__n__iter__tdsflowreate < 100");
}
```

The function `flowrateModel` is loaded from the macro `UserFunctions.C` (the file can be found in `${URANIESYS}/`

```
gROOT->LoadMacro("UserFunctions.C");
```

Each parameter is related to the `TDataServer` as a `TAttribute` and obeys an uniform law on specific interval:

```
TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

To instantiate the `TSobol`, one uses the `TDataServer`, the name of the function and the number of samplings needed to perform sensitivity analysis (here ns=600):

```
TSobol * tsobol = new TSobol(tds, "flowrateModel", ns, "rw:r:tu:tl:hu:hl:l:kw", " ↩
    flowrateModel", "pouet");
```

Computation of the sensitivity indexes:

```
tsobol->computeIndexes();
```

The automatic backup of data (the file `_sobol_launching_.dat`) is renamed so that it can be used in other macros (see Section XIV.5.13 and Section XIV.5.15) while the tds contains is exported (only the M and N matrices content) to be also used in another macro (Section XIV.5.14) :

```
gSystem->Rename("_sobol_launching_.dat","ref_sobol_launching_.dat");
tds->exportData("_onlyMandN_sobol_launching_.dat","rw:r:tu:tl:hu:hl:l:kw:flowrateModel"," ↩
    sobol__n__iter__tdsflowreate < 100");
```

XIV.5.9.3 Graph

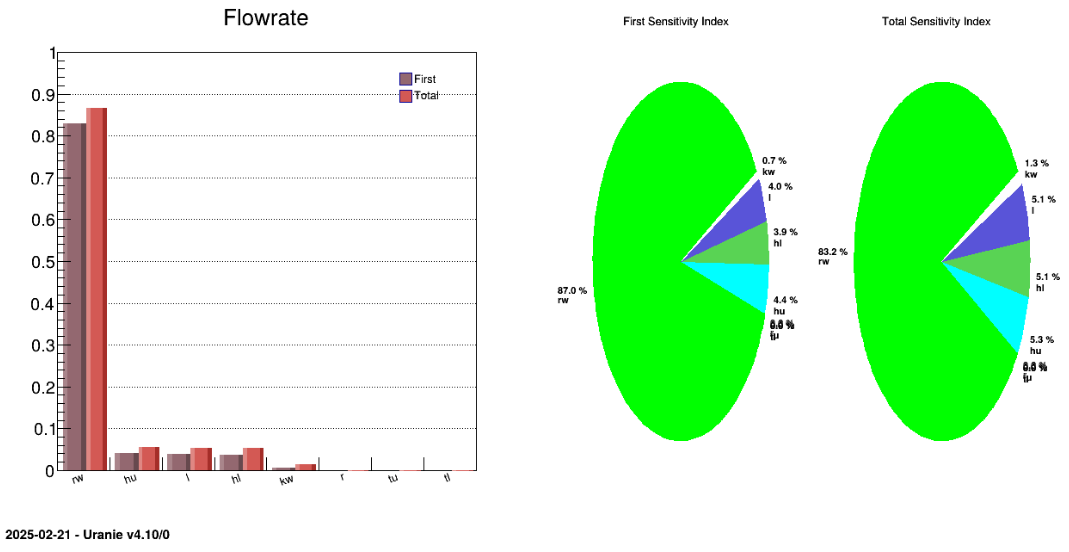


Figure XIV.51: Graph of the macro "sensitivitySobolFunctionFlowrate.C"

XIV.5.9.4 Console

```
Processing sensitivitySobolFunctionFlowrate.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8530]
 <URANIE::WARNING> TDataServer::getTuple Error : There is no tree!
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${SOURCEDIR}/meTIER/sampler/souRCE/TSamplerStochastic.cxx] Line ↩
     [66]
 <URANIE::INFO> TSamplerStochastic::init: the TDS [tdsflowreate] contains data: we need to  ↩
     empty it !
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>
 ** Case of Output atty [flowrateModel]  nSimPerIndex 10000
 ** Input att [rw] First [0.830033] Total Order[0.865762]
 ** Input att [r] First [0] Total Order[0.000102212]
 ** Input att [tu] First [0] Total Order[0.0001]
 ** Input att [tl] First [0] Total Order[0.000110756]
 ** Input att [hu] First [0.0417298] Total Order[0.0554922]
 ** Input att [hl] First [0.0367345] Total Order[0.0526188]
 ** Input att [l] First [0.0384214] Total Order[0.0535728]
 ** Input att [kw] First [0.00669831] Total Order[0.0132316]
*********************************************************************************************************
```

```
*    Row   *   Out.Out *   Inp.Inp * Order.Ord * Method.Me * Algo.Algo * Value.Val *  ↩
   CILower.C * CIUpper.C *
*****************************************************************************************************

*        0 * flowrateM *        rw *     First *     Sobol * --first-- * 0.8300331 *  ↩
   0.8238356 * 0.8360321 *
*        4 * flowrateM *        rw *     Total *     Sobol * --total-- * 0.8657619 *  ↩
   0.8465652 * 0.8850599 *
*        8 * flowrateM *         r *     First *     Sobol * --first-- *         0 *  ↩
           0 * 0.0196004 *
*       12 * flowrateM *         r *     Total *     Sobol * --total-- * 0.0001022 * 9.828e ↩
   -05 * 0.0001062 *
*       16 * flowrateM *        tu *     First *     Sobol * --first-- *         0 *  ↩
           0 * 0.0196004 *
*       20 * flowrateM *        tu *     Total *     Sobol * --total-- * 0.0001000 * 9.615e ↩
   -05 * 0.0001039 *
*       24 * flowrateM *        tl *     First *     Sobol * --first-- *         0 *  ↩
           0 * 0.0196004 *
*       28 * flowrateM *        tl *     Total *     Sobol * --total-- * 0.0001107 *  ↩
   0.0001064 * 0.0001151 *
*       32 * flowrateM *        hu *     First *     Sobol * --first-- * 0.0417297 *  ↩
   0.0221474 * 0.0612800 *
*       36 * flowrateM *        hu *     Total *     Sobol * --total-- * 0.0554921 *  ↩
   0.0534156 * 0.0576470 *
*       40 * flowrateM *        hl *     First *     Sobol * --first-- * 0.0367345 *  ↩
   0.0171464 * 0.0562944 *
*       44 * flowrateM *        hl *     Total *     Sobol * --total-- * 0.0526187 *  ↩
   0.0506469 * 0.0546652 *
*       48 * flowrateM *         l *     First *     Sobol * --first-- * 0.0384214 *  ↩
   0.0188352 * 0.0579782 *
*       52 * flowrateM *         l *     Total *     Sobol * --total-- * 0.0535727 *  ↩
   0.0515661 * 0.0556552 *
*       56 * flowrateM *        kw *     First *     Sobol * --first-- * 0.0066983 *  ↩
           0 * 0.0262952 *
*       60 * flowrateM *        kw *     Total *     Sobol * --total-- * 0.0132315 *  ↩
   0.0127261 * 0.0137570 *
*       64 * flowrateM *   __sum__ *     First *     Sobol * --first-- * 0.9536171 *  ↩
          -1 *        -1 *
*       65 * flowrateM *   __sum__ *     Total *     Sobol * --total-- * 1.0409902 *  ↩
          -1 *        -1 *
*****************************************************************************************************

==> 18 selected entries
```

XIV.5.10 Macro "sensitivitySobolFunctionFlowrateRunner.C"

XIV.5.10.1 Objective

The objective of this macro is to perform Sobol sensitivity analysis on a set of eight parameters used in the flowrateModel model described in Section IV.1.2.1, but this time using the Relauncher architecture.

XIV.5.10.2 Macro Uranie

```
{

  gROOT->LoadMacro("UserFunctions.C");
  // Define the attributes
  TUniformDistribution rw("rw", 0.05, 0.15);
  TUniformDistribution r("r", 100.0, 50000.0);
  TUniformDistribution tu("tu", 63070.0, 115600.0);
  TUniformDistribution tl("tl", 63.1, 116.0);
  TUniformDistribution hu("hu", 990.0, 1110.0);
  TUniformDistribution hl("hl", 700.0, 820.0);
  TUniformDistribution l("l", 1120.0, 1680.0);
  TUniformDistribution kw("kw", 9855.0, 12045.0);

  // Create the evaluator
  TCIntEval code("flowrateModel");
  // Create output attribute
  TAttribute yout("flowrateModel");
  // Provide input/output attributes to the assessor
  code.setInputs(8, &rw, &r, &tu, &tl, &hu, &hl, &l, &kw);
  code.setOutputs(1, &yout);

  TSequentialRun run(&code); // To be replaced to distribute the computation
  run.startSlave();
  if( run.onMaster())
  {
     // Create the dataserver
     TDataServer *tds = new TDataServer("sobol", "foo bar pouet chocolat");
     tds->addAttribute(&rw);
     tds->addAttribute(&r);
     tds->addAttribute(&tu);
     tds->addAttribute(&tl);
     tds->addAttribute(&hu);
     tds->addAttribute(&hl);
     tds->addAttribute(&l);
     tds->addAttribute(&kw);

     // Create the sobol object
     Int_t ns = 100000;
     TSobol * tsobol = new TSobol(tds, &run, ns);
     tsobol->setDrawProgressBar(kFALSE);
     tsobol->computeIndexes();

     if ( 1 ) {
       TCanvas *cc = new TCanvas("c1", "histgramme",5,64,1270,667);
       TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
       pad->Divide(2,1);
       pad->cd(1);
       tsobol->drawIndexes("Flowrate", "", "nonewcanv,hist,all");

       pad->cd(2);
       tsobol->drawIndexes("Flowrate", "", "nonewcanv,pie,first");

    }
  }
}
```

The function `flowrateModel` is loaded from the macro `UserFunctions.C` (the file can be found in `${URANIESYS}/share`

```
gROOT->LoadMacro("UserFunctions.C");
```

Each parameter is related to the `TDataServer` as a `TAttribute` and obeys an uniform law on specific interval:

```
// Define the attributes
TUniformDistribution rw("rw", 0.05, 0.15);
TUniformDistribution r("r", 100.0, 50000.0);
TUniformDistribution tu("tu", 63070.0, 115600.0);
TUniformDistribution tl("tl", 63.1, 116.0);
TUniformDistribution hu("hu", 990.0, 1110.0);
TUniformDistribution hl("hl", 700.0, 820.0);
TUniformDistribution l("l", 1120.0, 1680.0);
TUniformDistribution kw("kw", 9855.0, 12045.0);
```

The interface to the function is then defined, using the Relauncher interface, through a `TCIntEval` object and a sequential runner:

```
 // Create the evaluator
TCIntEval code("flowrateModel");
// Create output attribute
TAttribute yout("flowrateModel");
// Provide input/output attributes to the assessor
code.setInputs(8, &rw, &r, &tu, &tl, &hu, &hl, &l, &kw);
code.setOutputs(1, &yout);

TSequentialRun run(&code); // To be replaced to distribute the computation
run.startSlave();
```

To instantiate the `TSobol`, one uses the `TDataServer`, a pointer to the runner and the number of samplings needed to perform sensitivity analysis (here ns=600):

```
TSobol * tsobol = new TSobol(tds, &run, ns);
```

Computation of the sensitivity indexes:

```
tsobol->computeIndexes();
```

Data are exported from the `TDataServer` to an ASCII file:

```
tds->exportData("_sobol_launching_.dat");
```

XIV.5.10.3 Graph

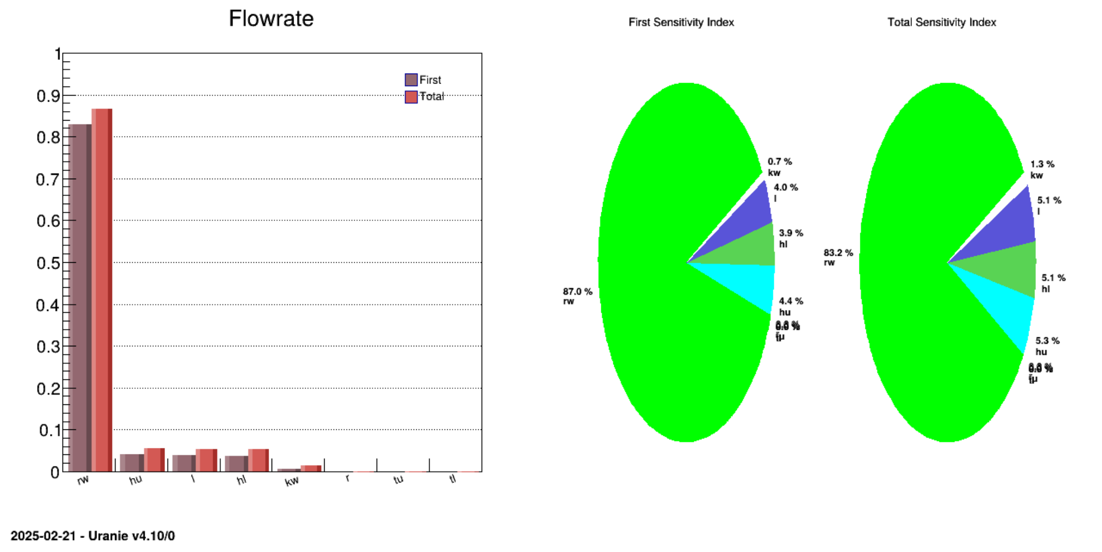


Figure XIV.52: Graph of the macro "`sensitivitySobolFunctionFlowrateRunner.C`"

XIV.5.10.4 Console

```
Processing sensitivitySobolFunctionFlowrateRunner.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8530]
 <URANIE::WARNING> TDataServer::getTuple Error : There is no tree!
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${SOURCEDIR}/meTIER/sampler/souRCE/TSamplerStochastic.cxx] Line ↩
     [66]
 <URANIE::INFO> TSamplerStochastic::init: the TDS [sobol] contains data: we need to empty  ↩
     it !
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>
 ** Case of Output atty [flowrateModel]  nSimPerIndex 10000
 ** Input att [rw] First [0.830033] Total Order[0.865762]
 ** Input att [r] First [0] Total Order[0.000102212]
 ** Input att [tu] First [0] Total Order[0.0001]
 ** Input att [tl] First [0] Total Order[0.000110756]
 ** Input att [hu] First [0.0417298] Total Order[0.0554922]
 ** Input att [hl] First [0.0367345] Total Order[0.0526188]
 ** Input att [l] First [0.0384214] Total Order[0.0535728]
 ** Input att [kw] First [0.00669831] Total Order[0.0132316]
```

XIV.5.11 Macro "sensitivityRegressionLeveLE.C"

Warning
The **levele** command will be installed on your machine only if a Fortran compiler is found

XIV.5.11.1 Objective

The objective of this macro is to perform a SRC and SRRC measurement on the temporal use-case **levele**. This use-case is an example of code that takes a dozen of entries in order to compute the evolution of dose as a function of time. The result of every computation consists in 3 vectors: the time (always the same value disregarding all entries), the dose called "y" and a third useless information.

XIV.5.11.2 Macro Uranie

```
{
  // OS abstraction
  string which_levele =
    string(gSystem->GetBuildArch()) == "win64" ? "where levele" : "which levele";
```

```
//Exit if levele not found
if(gSystem->Exec(which_levele.c_str()))
  exit(-1);

//Create DataServer and add input attributes
TDataServer *tds = new URANIE::DataServer::TDataServer("tds", "levelE usecase");
tds->addAttribute( new TUniformDistribution("t", 100, 1000));
tds->addAttribute( new TLogUniformDistribution("kl", 0.001, .01));
tds->addAttribute( new TLogUniformDistribution("kc", 1.0e-6, 1.0e-5));
tds->addAttribute( new TLogUniformDistribution("v1", 1.0e-3, 1.0e-1));
tds->addAttribute( new TUniformDistribution("l1", 100., 500.));
tds->addAttribute( new TUniformDistribution("r1", 1., 5.));
tds->addAttribute( new TUniformDistribution("rc1", 3., 30.));
tds->addAttribute( new TLogUniformDistribution("v2", 1.0e-2, 1.0e-1));
tds->addAttribute( new TUniformDistribution("l2", 50., 200.));
tds->addAttribute( new TUniformDistribution("r2", 1., 5.));
tds->addAttribute( new TUniformDistribution("rc2", 3., 30.));
tds->addAttribute( new TLogUniformDistribution("w", 1.0e5, 1.0e7));

//Tell the code where to find attribute value in input file
TString sIn = "levelE_input_with_values_rows.in";
tds->getAttribute("t")->setFileKey(sIn, "", "%e",TAttributeFileKey::kNewRow);
tds->getAttribute("kl")->setFileKey(sIn,"", "%e",TAttributeFileKey::kNewRow);
tds->getAttribute("kc")->setFileKey(sIn, "", "%e",TAttributeFileKey::kNewRow);
tds->getAttribute("v1")->setFileKey(sIn, "", "%e",TAttributeFileKey::kNewRow);
tds->getAttribute("l1")->setFileKey(sIn, "", "%e",TAttributeFileKey::kNewRow);
tds->getAttribute("r1")->setFileKey(sIn, "", "%e",TAttributeFileKey::kNewRow);
tds->getAttribute("rc1")->setFileKey(sIn, "", "%e",TAttributeFileKey::kNewRow);
tds->getAttribute("v2")->setFileKey(sIn, "", "%e",TAttributeFileKey::kNewRow);
tds->getAttribute("l2")->setFileKey(sIn, "", "%e",TAttributeFileKey::kNewRow);
tds->getAttribute("r2")->setFileKey(sIn, "", "%e",TAttributeFileKey::kNewRow);
tds->getAttribute("rc2")->setFileKey(sIn, "", "%e",TAttributeFileKey::kNewRow);
tds->getAttribute("w")->setFileKey(sIn, "", "%e",TAttributeFileKey::kNewRow);

// Create DOE
Int_t ns = 1024;
TSampling *samp = new TSampling(tds, "lhs", ns);
samp->generateSample();

//How to read ouput files
TOutputFileRow *out = new TOutputFileRow("_output_levelE_withRow_.dat");
//Tell the output file that attribute IS a vector and is SECOND column
out->addAttribute(new TAttribute("y", TAttribute::kVector), 2 );

//Creation of TCode
TCode *myc = new TCode(tds," levele 2> /dev/null");
myc->addOutputFile(out);

//Run the code
TLauncher * tl = new TLauncher(tds, myc);
tl->run();

//Launch Regression
TRegression *tsen = new TRegression(tds, "t:kl:kc:v1:l1:r1:rc1:v2:l2:r2:rc2:w" , "y", " ↩
    SRCSRRC");
tsen->computeIndexes();
TTree *res=tsen->getResultTuple();

//Plotting mess
double tps[26]={20000,30000,40000,50000,60000,70000,80000,90000,100000,  ↩
    200000,300000,400000,500000,600000,700000,800000,900000, 1e+06,2e+06,3e+06,4e+06,5e ↩
    +06,6e+06,7e+06,8e+06,9e+06};
int colors[12] ={1,2,3,4,6,7,8,15,30,38,41,46};

TCanvas *c2 = new TCanvas("c2","c2",5,64,1600,500);
TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
pad->Divide(3,1); pad->cd(1);
gPad->SetLogx(); gPad->SetGrid();
TMultiGraph *mg = new TMultiGraph();
res->Draw("Value","Inp==\"__R2__\" && Order==\"Total\" && Method==\"SRRC^2\"","goff");
double *data3=res->GetV1();

TGraph *gr3 = new TGraph(26,tps,data3); gr3->SetMarkerColor(2); gr3->SetLineColor(2); gr3 ↩
    ->SetMarkerStyle(23);
mg->Add(gr3);
res->Draw("Value","Inp==\"__R2__\" && Order==\"Total\" && Method==\"SRC^2\"","goff");
double *data4=res->GetV1();

TGraph *gr4 = new TGraph(26,tps,data4); gr4->SetMarkerColor(4); gr4->SetLineColor(4); gr4 ↩
    ->SetMarkerStyle(23);
mg->Add(gr4);
mg->Draw("APC");
mg->GetXaxis()->SetTitle("Time");
mg->GetYaxis()->SetTitle("#sum Sobol");
mg->GetYaxis()->SetRangeUser(0.0,1.0);

//Legend
gStyle->SetLegendBorderSize(0);
gStyle->SetFillStyle(0);

TLegend *lg = new TLegend(0.25,0.7,0.45,0.9);
lg->AddEntry(gr4,"R2 SRC","lp");
lg->AddEntry(gr3,"R2 SRRC","lp");
lg->Draw();


pad->cd(2);  gPad->SetLogx();  gPad->SetGrid();
TMultiGraph *mg2 = new TMultiGraph();

string names[12] = {"t","kl","kc","v1","l1","r1","rc1","v2","l2","r2","rc2","w"};
TGraph *src[12];
TLegend *leg = new TLegend(0.25,0.3,0.45,0.89,"Cumulative contributions");
leg->SetTextSize(0.035);
for(unsigned int igr=0; igr<12; igr++)
{
  string sel="Inp==\""+names[igr]+"\" && Order==\"Total\" && Method==\"SRC^2\" && Algo ↩
      !=\"--rho^2--\"";
  res->Draw("Value", sel.c_str(),"goff");
  double *data=res->GetV1();
  src[igr] = new TGraph();
  src[igr]->SetMarkerColor(colors[igr]); src[igr]->SetLineColor(colors[igr]);  src[igr]-> ↩
      SetFillColor(colors[igr]);

  src[igr]->SetPoint(0, 0.99999999*tps[0], 0);
  for(unsigned int ip=0; ip<26; ip++)
  {
    double x=0, y=0;
    if(igr!=0)
      src[igr-1]->GetPoint(ip+1,x,y);
    src[igr]->SetPoint(ip+1, tps[ip], y+data[ip]);
  }
  src[igr]->SetPoint(27, tps[25]*1.000000001, 0);
  leg->AddEntry(src[igr],names[igr].c_str(),"f");
}
for(int igr2=11; igr2>-1; igr2--)
  mg2->Add(src[igr2]);

mg2->Draw("AFL");
mg2->GetXaxis()->SetTitle("Time");
mg2->GetYaxis()->SetTitle("SRC^{2}");
mg2->GetYaxis()->SetRangeUser(0.0,0.3);
leg->Draw();

pad->cd(3); gPad->SetLogx(); gPad->SetGrid();
TMultiGraph *mg3 = new TMultiGraph();
TGraph *srrc[12];
for(unsigned int igr=0; igr<12; igr++)
{
  string sel="Inp==\""+names[igr]+"\" && Order==\"Total\" && Method==\"SRRC^2\" && Algo ↩
      !=\"--rho^2--\"";
  res->Draw("Value", sel.c_str(),"goff");
  double *data=res->GetV1();
  srrc[igr] = new TGraph();
  srrc[igr]->SetMarkerColor(colors[igr]); srrc[igr]->SetLineColor(colors[igr]);  srrc[igr ↩
      ]->SetFillColor(colors[igr]);

  srrc[igr]->SetPoint(0, 0.99999999*tps[0], 0);
  for(unsigned int ip=0; ip<26; ip++)
  {
    double x=0, y=0;
    if(igr!=0)
      srrc[igr-1]->GetPoint(ip+1,x,y);
    srrc[igr]->SetPoint(ip+1, tps[ip], y+data[ip]);
  }
  srrc[igr]->SetPoint(27, tps[25]*1.000000001, 0);
  srrc[igr]->SetTitle( names[igr].c_str() );
}

for(int igr2=11; igr2>-1; igr2--)
  mg3->Add(srrc[igr2]);

//  mg3->Draw("a fb l3d");
mg3->Draw("AFL");
mg3->GetXaxis()->SetTitle("Time");
mg3->GetYaxis()->SetTitle("SRRC^{2}");
mg3->GetYaxis()->SetRangeUser(0.0,1.0);
leg->Draw();

}
```

The **levele** external code is located in the bin directory of the Uranie installation.

When looking at the code and comparing it to an usual Regression estimation, the organisation is completely transparent. The only noticeable (and compulsory) thing to do is to change the default type of the attribute read at the end of the job. This is done in this line:

```
out->addAttribute(new TAttribute("y", TAttribute::kVector), 2 );
```

where the output attribute is provided, changing its nature to a vector, thanks to the second argument of the `TAttribute` constructor from the default (`kReal`) to the desired nature (`kVector`). Once this is done, this information is broadcast internally to the code that knows how to deal with this type of attribute.

The rest of the code is the graphical part, leading to the figure below (it is provided to illustrate how to represent results).

XIV.5.11.3 Graph

The results of the previous macro is shown in Figure XIV.53, where the left panel represents the value of the R^2 coefficients both the SRC and SRRC coefficients estimation. The middle and right panel display the cumulative sum of the quadratic value of the coefficient respectively for the SRC and SRRC case.

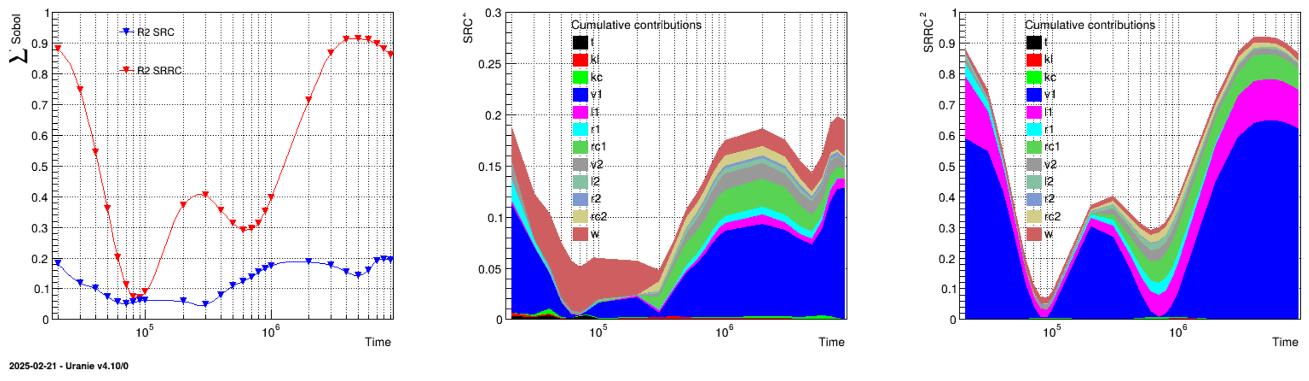


Figure XIV.53: Graph of the macro "sensitivityRegressionLeveLE.C"

XIV.5.12 Macro "sensitivitySobolLeveLE.C"

Warning
The **levele** command will be installed on your machine only if a Fortran compiler is found

XIV.5.12.1 Objective

The objective of this macro is to perform a full Sobol analysis on the temporal use-case **levele**. This use-case is an example of code that takes a dozen of entries in order to compute the evolution of dose as a function of time. The result of every computation consists in 3 vectors: the time (always the same value disregarding all entries), the dose called "y" and a third useless information.

XIV.5.12.2 Macro Uranie

```
{
  // OS abstraction
  string which_levele =
    string(gSystem->GetBuildArch()) == "win64" ? "where levele" : "which levele";

  //Exit if levele not found
  if(gSystem->Exec(which_levele.c_str()))
    exit(-1);

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsLevelE","levele");
  tds->addAttribute( new TUniformDistribution("t", 100, 1000) );
  tds->addAttribute( new TLogUniformDistribution("kl", 0.001, 0.01) );
  tds->addAttribute( new TLogUniformDistribution("kc", 0.000001, 0.00001) );
  tds->addAttribute( new TLogUniformDistribution("v1", 0.001, 0.1) );
  tds->addAttribute( new TUniformDistribution("l1", 100, 500) );
  tds->addAttribute( new TUniformDistribution("r1", 1, 5) );
  tds->addAttribute( new TUniformDistribution("rc1", 3, 30) );
  tds->addAttribute( new TLogUniformDistribution("v2", 0.01, 0.1) );
  tds->addAttribute( new TUniformDistribution("l2", 50, 200) );
  tds->addAttribute( new TUniformDistribution("r2", 1, 5) );
  tds->addAttribute( new TUniformDistribution("rc2", 3, 30) );
  tds->addAttribute( new TLogUniformDistribution("w", 100000, 10000000) );

  //Tell the code where to find attribute value in input file
  TString sIn="levelE_input_with_values_rows.in";
  tds->getAttribute("t")->setFileKey(sIn, "", "%e", TAttributeFileKey::kNewRow);
  tds->getAttribute("kl")->setFileKey(sIn, "", "%e", TAttributeFileKey::kNewRow);
  tds->getAttribute("kc")->setFileKey(sIn, "", "%e", TAttributeFileKey::kNewRow);
  tds->getAttribute("v1")->setFileKey(sIn, "", "%e", TAttributeFileKey::kNewRow);
  tds->getAttribute("l1")->setFileKey(sIn, "", "%e", TAttributeFileKey::kNewRow);
  tds->getAttribute("r1")->setFileKey(sIn, "", "%e", TAttributeFileKey::kNewRow);
  tds->getAttribute("rc1")->setFileKey(sIn, "", "%e", TAttributeFileKey::kNewRow);
  tds->getAttribute("v2")->setFileKey(sIn, "", "%e", TAttributeFileKey::kNewRow);
  tds->getAttribute("l2")->setFileKey(sIn, "", "%e", TAttributeFileKey::kNewRow);
  tds->getAttribute("r2")->setFileKey(sIn, "", "%e", TAttributeFileKey::kNewRow);
  tds->getAttribute("rc2")->setFileKey(sIn, "", "%e", TAttributeFileKey::kNewRow);
  tds->getAttribute("w")->setFileKey(sIn, "", "%e", TAttributeFileKey::kNewRow);

  //How to read ouput files
  TOutputFileRow *out = new TOutputFileRow("_output_levelE_withRow_.dat");
  //Tell the output file that attribute IS a vector and is SECOND column
  out->addAttribute(new TAttribute("y", TAttribute::kVector), 2 );

  //Creation of TCode
  TCode *myc = new TCode(tds,"levele 2> /dev/null");
  myc->addOutputFile(out);

  //Run Sobol analysis
  TSobol * tsobol = new TSobol(tds, myc, 10000);
  tsobol->computeIndexes();

  TTree *ntresu = (TTree*)tsobol->getResultTuple();

  //Plotting mess
  int colors[12] ={1,2,3,4,6,7,8,15,30,38,41,46};
  double tps[26]={20000,30000,40000,50000,60000,70000,80000,90000,100000,  ↩
      200000,300000,400000,500000,600000,700000,800000,900000, 1e+06,2e+06,3e+06,4e+06,5e ↩
      +06,6e+06,7e+06,8e+06,9e+06};

  TCanvas *c2 = new TCanvas("c2","c2",5,64,1200,900);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(1,2); pad->cd(1); gPad->SetLogx();  gPad->SetGrid();

  //LegendandMArker
  gStyle->SetMarkerStyle(3);
  gStyle->SetLegendBorderSize(0);
  gStyle->SetFillStyle(0);

  TMultiGraph *mg2 = new TMultiGraph();
  string names[12] = {"t","kl","kc","v1","l1","r1","rc1","v2","l2","r2","rc2","w"};
  TGraphAsymmErrors *fdeg[12];
  TLegend *leg[6]; double *data, *themin, *themax;

  for(unsigned int igr=0; igr<12; igr++)
  {
```

```
  stringstream sel; sel << "Inp==\"" << names[igr] << "\" && Order==\"First\" && Algo ↩
      ==\"martinez11\"";
  ntresu->Draw("CILower:CIUpper:Value", sel.str().c_str(),"goff");
  data=ntresu->GetV3();
  themin=ntresu->GetV1();
  themax=ntresu->GetV2();
  for(unsigned int i=0; i<26; i++)
    {
themin[i] = data[i] - themin[i];
themax[i] = - data[i] + themax[i];
    }

  if(igr%2==0)
    {
leg[igr/2] = new TLegend(0.1+0.15*(igr/2),0.91,0.25+0.15*(igr/2),0.98);
leg[igr/2]->SetTextSize(0.045);
    }

  fdeg[igr] = new TGraphAsymmErrors(26,tps, data,0, 0, themin, themax);
  fdeg[igr]->SetMarkerColor(colors[igr]); fdeg[igr]->SetLineColor(colors[igr]);  fdeg[igr ↩
      ]->SetFillColor(colors[igr]);
  leg[igr/2]->AddEntry(fdeg[igr],names[igr].c_str(),"pl");
}

for(int igr2=11; igr2>-1; igr2--)
  mg2->Add(fdeg[igr2]);

mg2->Draw("APC");
mg2->GetXaxis()->SetTitle("Time");
mg2->GetYaxis()->SetTitle("S_{1}[martinez11]");
for( int igr3=0; igr3<6; igr3++ )
  leg[igr3]->Draw();


TMultiGraph *mg = new TMultiGraph();
TGraphAsymmErrors *tdeg[12];
pad->cd(2); gPad->SetLogx(); gPad->SetGrid();
for(unsigned int igr=0; igr<12; igr++)
{
  stringstream sel; sel << "Inp==\"" << names[igr] << "\" && Order==\"Total\" && Algo ↩
      ==\"martinez11\"";
  ntresu->Draw("CILower:CIUpper:Value", sel.str().c_str(),"goff");
  data=ntresu->GetV3();
  themin=ntresu->GetV1();
  themax=ntresu->GetV2();
  for(unsigned int i=0; i<26; i++)
    {
themin[i] = data[i] - themin[i];
themax[i] = - data[i] + themax[i];
    }


  for(unsigned int ip=0; ip<26; ip++)
    {
  if(ip==0)
    {
      tdeg[igr] = new TGraphAsymmErrors();
      tdeg[igr]->SetMarkerColor(colors[igr]); tdeg[igr]->SetLineColor(colors[igr]);  tdeg ↩
          [igr]->SetFillColor(colors[igr]);
    }

  tdeg[igr]->SetPoint(ip, tps[ip], data[ip]);
  tdeg[igr]->SetPointError(ip, 0, 0, themin[ip], themax[ip]);

      }
  }

  for(int igr2=11; igr2>-1; igr2--)
    mg->Add(tdeg[igr2]);

  mg->Draw("APC");
  mg->GetXaxis()->SetTitle("Time");
  mg->GetYaxis()->SetTitle("S_{T}[martinez11]");
  for( int igr3=0; igr3<6; igr3++ )
    leg[igr3]->Draw();

}
```

The **levele** external code is located in the bin directory of the Uranie installation.

When looking at the code and comparing it to an usual Sobol estimation, the organisation is completely transparent. The only noticeable (and compulsory) thing to do is to change the default type of the attribute read at the end of the job. This is done in this line:

```
out->addAttribute(new TAttribute("y", TAttribute::kVector), 2 );
```

where the output attribute is provided, changing its nature to a vector, thanks to the second argument of the `TAttribute` constructor from the default (`kReal`) to the desired nature (`kVector`). Once this is done, this information is broadcast internally to the code that knows how to deal with this type of attribute.

The rest of the code is the graphical part, leading to the figure below (it is provided to illustrate how to represent results).

XIV.5.12.3 Graph

The results of the previous macro is shown in Figure XIV.54, where the evolution of the sobol coefficient is shown for all inputs with the uncertainty band, for the first order coefficient and the total one, respectively on the top and bottom panel.

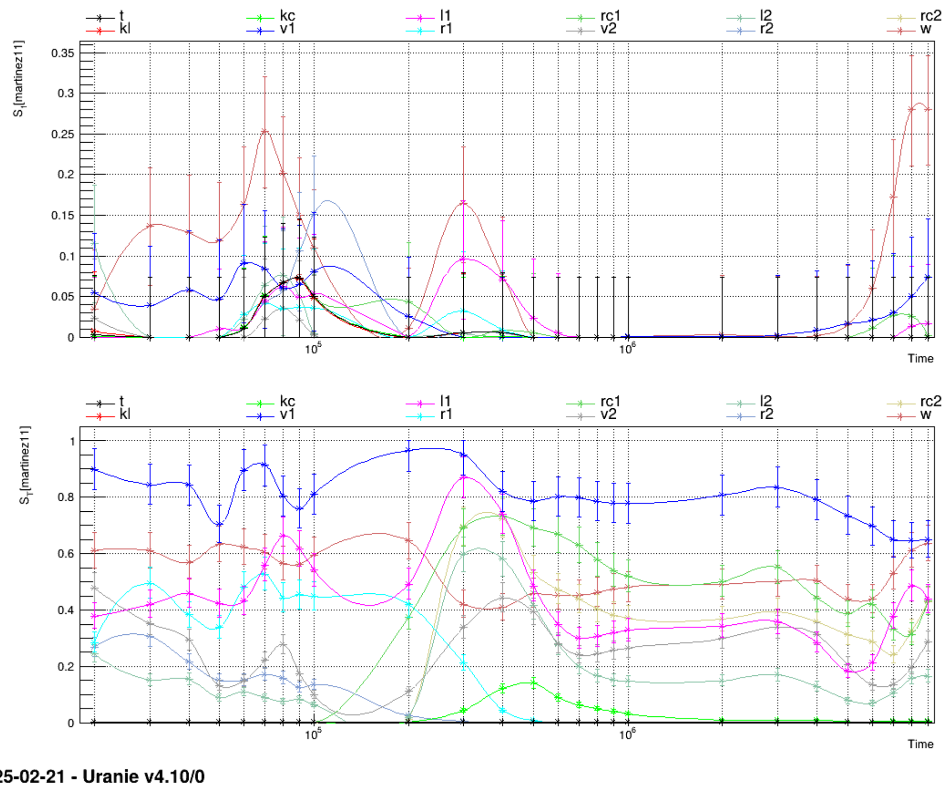


Figure XIV.54: Graph of the macro "`sensitivitySobolLeveLE.C`"

XIV.5.13 Macro "`sensitivitySobolRe-estimation.C`"

XIV.5.13.1 Objective

The objective of this macro is to perform a full Sobol analysis using the existing file created when the Sobol class is allowed to perform the design-of-experiments and the estimations by itself (see the first item in the tip box in Section VI.5.2 for more details). This would mean that the only computation done would be to estimate the coefficients (no external code / function called).

Warning The `ref_sobol_launching_.dat` file used as input is not provided in the usual sub-directory *"/share/uranie/macros"* of the installation folder of Uranie (*$URANIESYS*) but can be generated by the user by running the macro discussed in Section XIV.5.9.

XIV.5.13.2 Macro Uranie

```
{

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
  tds->fileDataRead("ref_sobol_launching_.dat");

  TSobol * tsobol = new TSobol(tds, "rw:r:tu:tl:hu:hl:l:kw", "flowrateModel");
  tsobol->computeIndexes();
  TCanvas *cc = new TCanvas("c1", "histgramme",5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2,1);
  pad->cd(1);
  tsobol->drawIndexes("Flowrate", "", "nonewcanv,hist,all");

  pad->cd(2);
  tsobol->drawIndexes("Flowrate", "", "nonewcanv,pie,first");

}
```

There are no external code or function to be run here. The input file `ref_sobol_launching_.dat` has to be generated by the use of `sensitivtySobolFunctionFlowrate.C`. Once done it is loaded into the dataserver and the `TSobol` object is constructed from the simplest constructor with only the pointer to the dataserver, the input and output list:

```
TSobol * tsobol = new TSobol(tds, "rw:r:tu:tl:hu:hl:l:kw", "flowrateModel");
```

Once done, the `computeIndexes()` method is called and few lines are shown to display the results in the classical plot form, leading to the figure below (it is provided to illustrate how to represent results). The numerical results are shown in the console below and are identical to the ones shown in Section XIV.5.9.4 from where the full sample is coming from.

XIV.5.13.3 Graph

The results of the previous macro is shown in Figure XIV.55, where the evolution of the sobol coefficient is shown for all inputs with the uncertainty band, for the first order coefficient and the total one, respectively on the top and bottom panel.

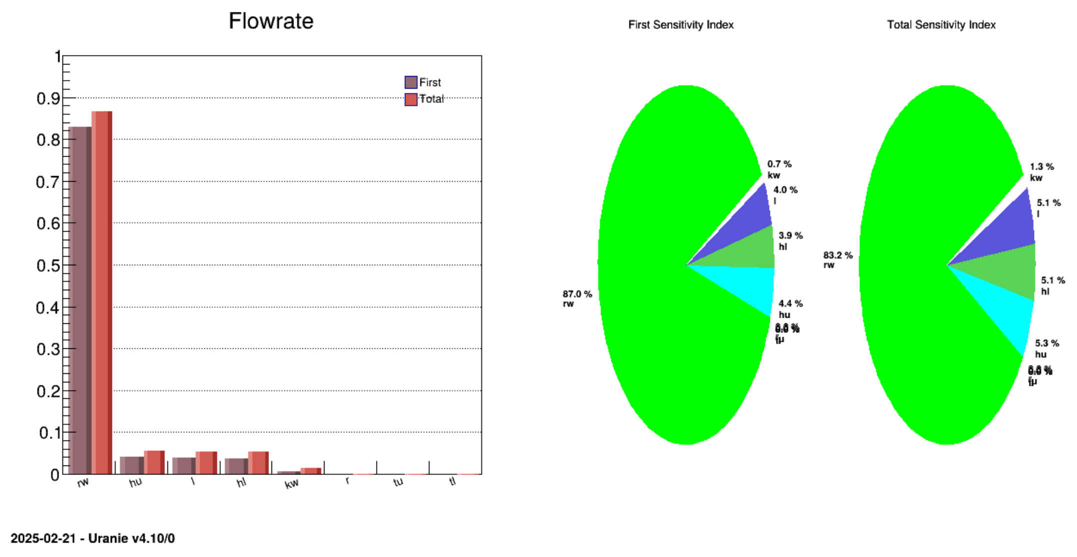


Figure XIV.55: Graph of the macro "`sensitivitySobolRe-estimation.C`"

XIV.5.13.4 Console

```
Processing sensitivitySobolRe-estimation.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 ** Case of Output atty [flowrateModel]  nSimPerIndex 10000
 ** Input att [rw] First [0.830033] Total Order[0.865762]
 ** Input att [r] First [0] Total Order[0.000102212]
 ** Input att [tu] First [0] Total Order[0.0001]
 ** Input att [tl] First [0] Total Order[0.000110756]
 ** Input att [hu] First [0.0417298] Total Order[0.0554922]
 ** Input att [hl] First [0.0367345] Total Order[0.0526188]
 ** Input att [l] First [0.0384214] Total Order[0.0535728]
 ** Input att [kw] First [0.00669831] Total Order[0.0132316]
```

XIV.5.14 Macro "sensitivitySobolWithData.C"

XIV.5.14.1 Objective

The objective of this macro is to perform a Sobol analysis using the some already made computations in order to be able to save ressources. The idea (discussed in the second item in the tip box in Section VI.5.2 is indeed to use the provided points as the $2 \times n_S$ first estimations corresponding to both the M and N matrices content. The class will still have to create all the cross configurations (the N_i matrices) and launch their corresponding estimations. In order to do that there are few things to keep in mind:

- The input file (here `_onlyMandN_sobol_launching_.dat`) should contains input and output variables only, the user being in charge of having a decent design-of-experiments for the sobol estimation.

Warning The `_onlyMandN_sobol_launching_.dat` file used as input is not provided in the usual sub-directory *"/share/uranie/macros"* of the installation folder of Uranie (*$URANIESYS*) but can be generated by the user by running the macro discussed in Section XIV.5.9.

XIV.5.14.2 Macro Uranie

```
{

  gROOT->LoadMacro("UserFunctions.C");

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
  tds->fileDataRead("_onlyMandN_sobol_launching_.dat");

  int ns=10000;
  TSobol * tsobol = new TSobol(tds, "flowrateModel", ns, "rw:r:tu:tl:hu:hl:l:kw", " ↩
      flowrateModel", "WithData");
  tsobol->setDrawProgressBar(kFALSE);
  tsobol->computeIndexes();

  TCanvas *cc = new TCanvas("c1", "histgramme",5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2,1);
  pad->cd(1);
  tsobol->drawIndexes("Flowrate", "", "nonewcanv,hist,all");

  pad->cd(2);
  tsobol->drawIndexes("Flowrate", "", "nonewcanv,pie,first");

}
```

As for Section XIV.5.9, the `UserFunctions.C` file is loaded and the input file `_onlyMandN_sobol_launching_.dat` (which should have been generated by the use of `sensitivtySobolFunctionFlowrate.C`), is loaded into the dataserver. The `TSobol` object is constructed from the usual function constructor with a noticing difference: the option field is filled with *WithData* to specify that data are already there and the code has to use these data and split them into both the M and N matrices.

```
TSobol * tsobol = new TSobol(tds, "flowrateModel", ns, "rw:r:tu:tl:hu:hl:l:kw", " ↩
    flowrateModel", "WithData");
```

Once done, the `computeIndexes()` method is called and few lines are shown to display the results in the classical plot form, leading to the figure below (it is provided to illustrate how to represent results). The numerical results are shown in the console below and are identical to the ones shown in Section XIV.5.9.4 from where the original set of points is coming from.

XIV.5.14.3 Graph

The results of the previous macro is shown in Figure XIV.56, where the evolution of the sobol coefficient is shown for all inputs with the uncertainty band, for the first order coefficient and the total one, respectively on the top and bottom panel.

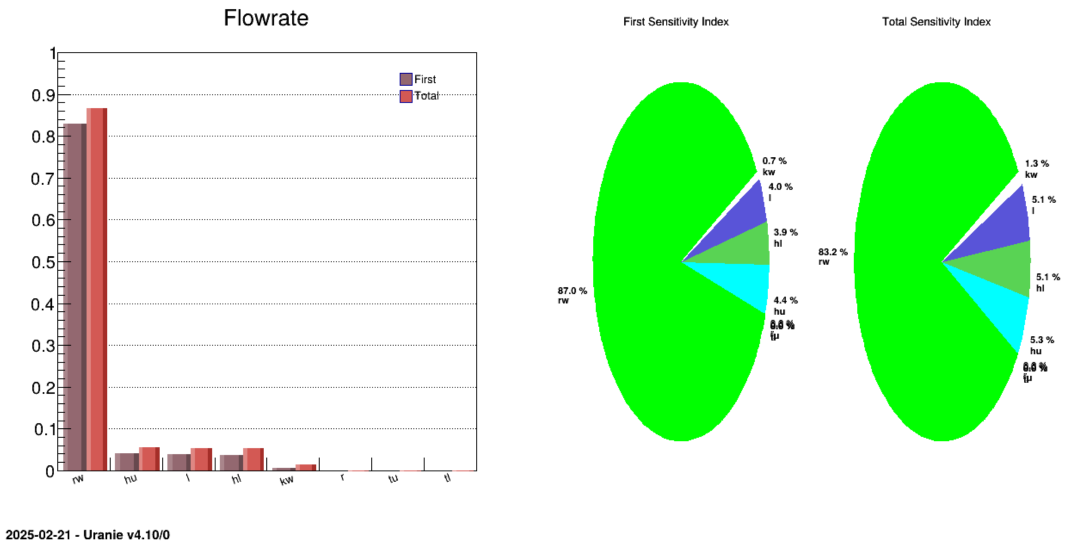


Figure XIV.56: Graph of the macro "`sensitivitySobolWithData.C`"

XIV.5.14.4 Console

```
Processing sensitivitySobolWithData.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 ** Case of Output atty [flowrateModel]  nSimPerIndex 10000
 ** Input att [rw] First [0.830033] Total Order[0.865762]
 ** Input att [r] First [0] Total Order[0.000102212]
 ** Input att [tu] First [0] Total Order[0.0001]
 ** Input att [tl] First [0] Total Order[0.000110756]
 ** Input att [hu] First [0.0417298] Total Order[0.0554922]
 ** Input att [hl] First [0.0367345] Total Order[0.0526188]
 ** Input att [l] First [0.0384214] Total Order[0.0535728]
 ** Input att [kw] First [0.00669831] Total Order[0.0132316]
```

XIV.5.15 Macro "sensitivitySobolLoadFile.C"

XIV.5.15.1 Objective

The objective of this macro is to perform a full Sobol analysis using the existing file created when the Sobol class is allowed to perform the design-of-experiments and the estimations by itself when this one is not considered accurate enough (from the statistical point of view). The idea is to use anyway all the computations already done and generate some more to increase the statistical precision. In order to do that there are few things to keep in mind:

- The problem should obviously be exactly the same: same input and output (name, statistical laws, parameter values...).
- The input file (here `ref_sobol_launching_.dat`) should contains the internal variable in order to figure out from what matrices every configuration is taken out of (generally an iterator whose name should look like `sobol__n__iter__` plus the dataserver name).
- If the pseudo-random generator seed is set to a given value, bBe sure that you are not be using the same seed than the once used to generate the input file, which would lead to twice the same events.

One can find another discussion for this objective in the third item in the tip box in Section VI.5.2.

Warning The `ref_sobol_launching_.dat` file used as input is not provided in the usual sub-directory *"/share/uranie/macros"* of the installation folder of Uranie (*$URANIESYS*) but can be generated by the user by running the macro discussed in Section XIV.5.9.

XIV.5.15.2 Macro Uranie

```
{

  gROOT->LoadMacro("UserFunctions.C");

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  int ns=100000;
  TSobol * tsobol = new TSobol(tds, "flowrateModel", ns, "rw:r:tu:tl:hu:hl:l:kw", " ↩
      flowrateModel");
  tsobol->loadOtherSobolFile("ref_sobol_launching_.dat");
  tsobol->setDrawProgressBar(kFALSE);
  tsobol->computeIndexes();

  tsobol->getResultTuple()->Scan("*","Algo==\"--first--\" || Algo==\"--total--\"");

  TCanvas *cc = new TCanvas("c1", "histgramme",5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2,1);
  pad->cd(1);
  tsobol->drawIndexes("Flowrate", "", "nonewcanv,hist,all");

  pad->cd(2);
  tsobol->drawIndexes("Flowrate", "", "nonewcanv,pie,first");

}
```

As for Section XIV.5.9, the `UserFunctions.C` file is loaded and all the input stochastic distribution are chosen wisely. The number of new estimations is set (to 10000, doubling the statistic as this is also the number used in Section XIV.5.9 which provided the input file that would be used to complete this estimation).

All the code lines are exactly those taken out of Section XIV.5.9.2 but the one that is used to load a previous estimation which is shown below:

```
tsobol->loadOtherSobolFile("ref_sobol_launching_.dat");
```

Once done, the `computeIndexes()` method is called and few lines are shown to display the results in the classical plot form, leading to the figure below (it is provided to illustrate how to represent results). The numerical results are shown in the console below and the improvement in terms of statistical precision can be seen by comparing the 95 percent confidence intervals going from Section XIV.5.9.4 to Section XIV.5.15.4.

XIV.5.15.3 Graph

The results of the previous macro is shown in Figure XIV.57, where the evolution of the sobol coefficient is shown for all inputs with the uncertainty band, for the first order coefficient and the total one, respectively on the top and bottom panel.

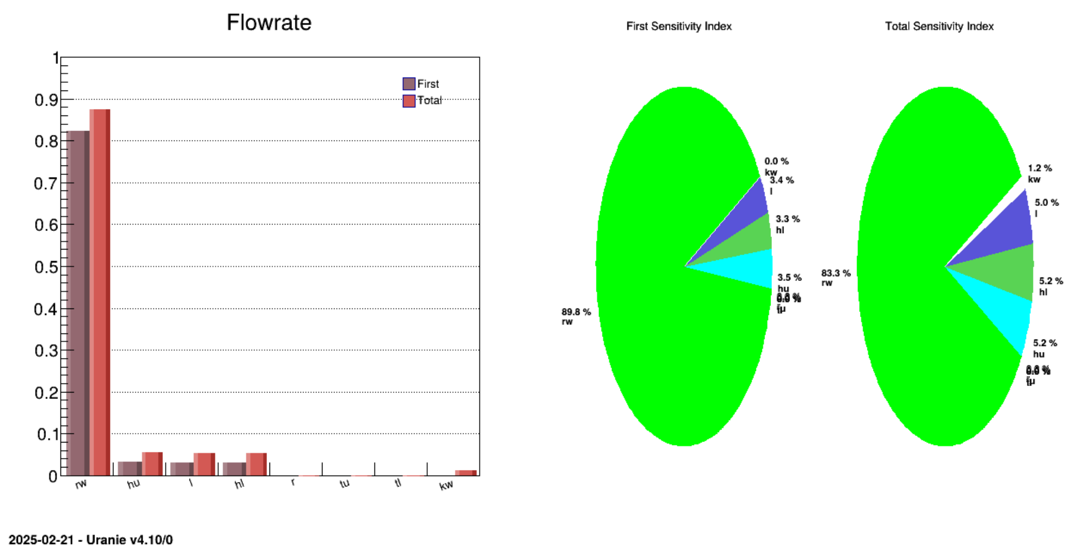


Figure XIV.57: Graph of the macro "`sensitivitySobolLoadFile.C`"

XIV.5.15.4 Console

```
Processing sensitivitySobolLoadFile.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                       Copyright (C) 2013-2025 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Fri Feb 21, 2025

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[759]
 <URANIE::WARNING> TDataServer::fileDataRead: Expected iterator tdsflowreate_1__n__iter__ ↩
     not found but tdsflowreate__n__iter__ looks like an URANIE iterator => Will be used as ↩
     so.
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8530]
 <URANIE::WARNING> TDataServer::getTuple Error : There is no tree!
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${SOURCEDIR}/meTIER/sampler/souRCE/TSamplerStochastic.cxx] Line ↩
     [66]
 <URANIE::INFO> TSamplerStochastic::init: the TDS [tdsflowreate] contains data: we need to ↩
     empty it !
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>
 ** Case of Output atty [flowrateModel]  nSimPerIndex 20000
 ** Input att [rw] First [0.823516] Total Order[0.873849]
 ** Input att [r] First [0] Total Order[5.22438e-05]
 ** Input att [tu] First [0] Total Order[5e-05]
 ** Input att [tl] First [0] Total Order[6.08391e-05]
 ** Input att [hu] First [0.0319889] Total Order[0.0547724]
 ** Input att [hl] First [0.0304453] Total Order[0.0540884]
 ** Input att [l] First [0.0315935] Total Order[0.0527791]
 ** Input att [kw] First [0] Total Order[0.0129065]
****************************************************************************************************

*    Row   *   Out.Out *   Inp.Inp * Order.Ord * Method.Me * Algo.Algo * Value.Val * ↩
    CILower.C * CIUpper.C *
****************************************************************************************************

*        0 * flowrateM *        rw *     First *     Sobol * --first-- * 0.8235162 * ↩
    0.8190045 * 0.8279262 *
*        4 * flowrateM *        rw *     Total *     Sobol * --total-- * 0.8738492 * ↩
    0.8602341 * 0.8875120 *
*        8 * flowrateM *         r *     First *     Sobol * --first-- *         0 * ↩
            0 * 0.0138594 *
*       12 * flowrateM *         r *     Total *     Sobol * --total-- * 5.224e-05 * 5.081e ↩
    -05 * 5.371e-05 *
*       16 * flowrateM *        tu *     First *     Sobol * --first-- *         0 * ↩
            0 * 0.0138594 *
*       20 * flowrateM *        tu *     Total *     Sobol * --total-- * 5.000e-05 * 4.863e ↩
    -05 * 5.140e-05 *
*       24 * flowrateM *        tl *     First *     Sobol * --first-- *         0 * ↩
            0 * 0.0138594 *
*       28 * flowrateM *        tl *     Total *     Sobol * --total-- * 6.083e-05 * 5.917e ↩
    -05 * 6.254e-05 *
*       32 * flowrateM *        hu *     First *     Sobol * --first-- * 0.0319888 * ↩
    0.0181374 * 0.0458280 *
*       36 * flowrateM *        hu *     Total *     Sobol * --total-- * 0.0547723 * ↩
    0.0533147 * 0.0562686 *
*       40 * flowrateM *        hl *     First *     Sobol * --first-- * 0.0304453 * ↩
    0.0165929 * 0.0442861 *
*       44 * flowrateM *        hl *     Total *     Sobol * --total-- * 0.0540884 * ↩
    0.0526485 * 0.0555665 *
*       48 * flowrateM *         l *     First *     Sobol * --first-- * 0.0315935 * ↩
    0.0177418 * 0.0454330 *
*       52 * flowrateM *         l *     Total *     Sobol * --total-- * 0.0527791 * ↩
    0.0513732 * 0.0542224 *
*       56 * flowrateM *        kw *     First *     Sobol * --first-- *         0 * ↩
            0 * 0.0138594 *
*       60 * flowrateM *        kw *     Total *     Sobol * --total-- * 0.0129065 * ↩
    0.0125558 * 0.0132669 *
*       64 * flowrateM *   __sum__ *     First *     Sobol * --first-- * 0.9175440 * ↩
           -1 *        -1 *
*       65 * flowrateM *   __sum__ *     Total *     Sobol * --total-- * 1.0485587 * ↩
           -1 *        -1 *
****************************************************************************************************

==> 18 selected entries
```

XIV.5.16 Macro "sensitivityJohnsonRWFunctionFlowrate.C"

XIV.5.16.1 Objective

The objective of this macro is to perform a sensitivity analysis using the Johnson's relative weight method on a set of eight parameters used in the `flowrateModel` model described in Section IV.1.2.1.

XIV.5.16.2 Macro Uranie

```
{
  gROOT->LoadMacro("UserFunctions.C");

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsflowrate", "DataBase flowrate");
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  // \param Size of a sampling.
  Int_t nS = 1000;
  TString FuncName="flowrateModel";

  TJohnsonRW *tjrw = new TJohnsonRW(tds, FuncName, nS, "rw:r:tu:tl:hu:hl:l:kw", FuncName);
  tjrw->computeIndexes();

  // Get the results on screen
  tjrw->getResultTuple()->Scan("Out:Inp:Method:Value","Order==\"First\"");

  // Get the results as plots
  TCanvas *cc = new TCanvas("canhist", "histgramme");
  tjrw->drawIndexes("Flowrate", "", "nonewcanv,hist,first");
  cc->Print("appliUranieFlowrateJohnsonRW1000Histogram.png");

  TCanvas *ccc = new TCanvas("canpie", "TPie");
  tjrw->drawIndexes("Flowrate", "", "nonewcanv,pie");
  ccc->Print("appliUranieFlowrateJohnsonRW1000Pie.png");

}
```

The function `flowrateModel` is loaded from the macro `UserFunctions.C` (the file can be found in `${URANIESYS}/share`

```
gROOT->LoadMacro("UserFunctions.C");
```

Each parameter is related to the `TDataServer` as a `TAttribute` and obeys an uniform law on specific interval

```
TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

To instantiate the `TJohnsonRW` object, one uses the `TDataServer`, the name of the function, the number of samplings needed to perform sensitivity analysis (here n_S=4000) and the input and output variables:

```
TJohnsonRW * tjrw = new TJohnsonRW(tds, FuncName, nS, "rw:r:tu:tl:hu:hl:l:kw", FuncName);
```

Computation of sensitivity indexes:

```
trbd->computeIndexes();
```

The rest is very common to all sensitivity macros discussed here: it will produce two plots (the first one being a histogram show below) and the console is also shown below for completness.

XIV.5.16.3 Graph

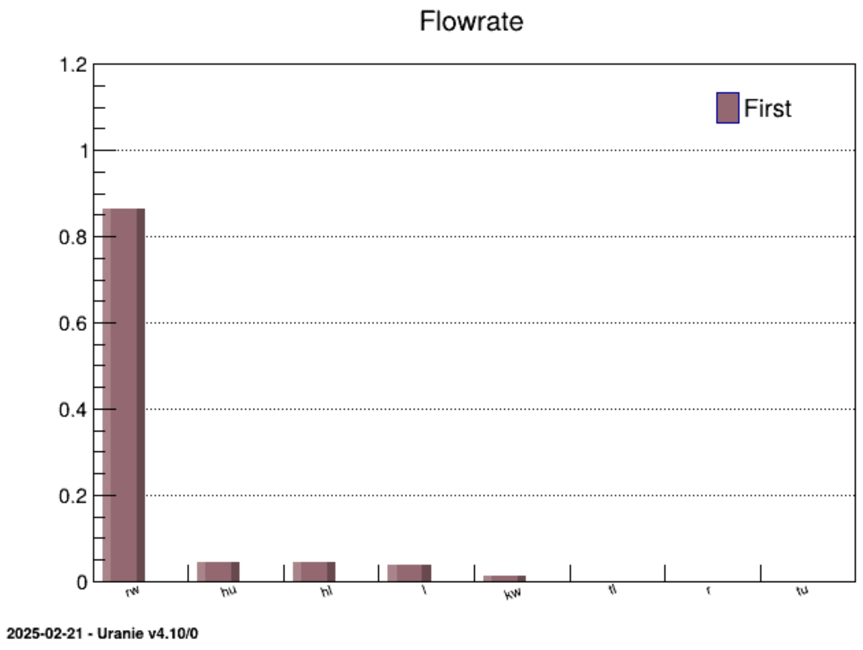


Figure XIV.58: Graph of the macro "sensitivityJohnsonRWFunctionFlowrate.C"

XIV.5.16.4 Console

```
Processing sensitivityJohnsonRWFunctionFlowrate.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8530]
 <URANIE::WARNING> TDataServer::getTuple Error : There is no tree!
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${SOURCEDIR}/meTIER/sampler/souRCE/TSamplerStochastic.cxx] Line ↩
     [66]
 <URANIE::INFO> TSamplerStochastic::init: the TDS [tdsflowrate] contains data: we need to  ↩
     empty it !
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>
************************************************************
*    Row   *       Out *       Inp *    Method *     Value *
************************************************************
*        0 * flowrateM *        rw * JohnsonRW * 0.8636266 *
*        2 * flowrateM *         r * JohnsonRW * 4.598e-05 *
*        4 * flowrateM *        tu * JohnsonRW * 8.775e-06 *
*        6 * flowrateM *        tl * JohnsonRW * 9.689e-05 *
*        8 * flowrateM *        hu * JohnsonRW * 0.0439173 *
*       10 * flowrateM *        hl * JohnsonRW * 0.0435594 *
*       12 * flowrateM *         l * JohnsonRW * 0.0378304 *
*       14 * flowrateM *        kw * JohnsonRW * 0.0109144 *
*       16 * flowrateM *    __R2__ * JohnsonRW * 0.9447092 *
*       18 * flowrateM *   __R2A__ * JohnsonRW * 0.9442633 *
************************************************************
==> 10 selected entries
```

XIV.5.17 Macro "sensitivityJohnsonRWCorrelatedFunctionFlowrate.C"

XIV.5.17.1 Objective

The objective of this macro is to perform a sensitivity analysis using the Johnson's relative weight method on a set of eight parameters used in the `flowrateModel` model described in Section IV.1.2.1. Compared to version detailled in Section XIV.5.16, the idea is here to correlate the input variables with a random correlation matrix.

XIV.5.17.2 Macro Uranie

```
// Function defined to generate randomly a good, highly-correlated, input correlation ↩
    matrix
TMatrixD GenCorr(int _nX=8, bool correlated=true)
{

    gRandom->SetSeed(__rdtsc());
    //Define a randomly filled matrix
    TMatrixD A(_nX,_nX);
    for (int i=0; i<_nX; i++)
        for (int j=0; j<_nX; j++)
            A(i,j)=gRandom->Gaus(0,1);

    // Compute AA^T and normalise it to get "covariance matrix"
    TMatrixD Gamma(A,TMatrixD::kMultTranspose,A);
    Gamma*=1./_nX;

    // Get the inverse of the diagonal matrix to do as if this was 1/sqrt(variance)
    TMatrixD Sig(_nX,_nX);
    for(int i=0;i<_nX;i++) Sig(i,i)=1./sqrt( Gamma(i,i) );

    // Compute the correlation matrix
    TMatrixD Corr(Sig*Gamma,TMatrixD::kMult, Sig);
    if( !correlated )
        Corr.UnitMatrix();

    return Corr;
}


void sensitivityJohnsonRWCorrelatedFunctionFlowrate()
{
  gROOT->LoadMacro("UserFunctions.C");

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsflowrate", "DataBase flowrate");
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  // \param Size of a sampling.
  Int_t nS = 1000;
  TString FuncName="flowrateModel";

  // Get a correlation matrix for the inputs
  TMatrixD inCorr=GenCorr(8,true);
  inCorr.Print();

  TJohnsonRW *tjrw = new TJohnsonRW(tds, FuncName, nS, "rw:r:tu:tl:hu:hl:l:kw", FuncName);
  //Set the correlation
  tjrw->setInputCorrelationMatrix(inCorr);
  tjrw->computeIndexes();

  // Get the results on screen
  tjrw->getResultTuple()->Scan("Out:Inp:Method:Value","Order==\"First\"");

  // Get the results as plots
  TCanvas *cc = new TCanvas("canhist", "histgramme");
  tjrw->drawIndexes("Flowrate", "", "nonewcanv,hist,first");
  cc->Print("appliUranieFlowrateJohnsonRWCorrelated1000Histogram.png");

  TCanvas *ccc = new TCanvas("canpie", "TPie");
  tjrw->drawIndexes("Flowrate", "", "nonewcanv,pie");
  ccc->Print("appliUranieFlowrateJohnsonRWCorrelated1000Pie.png");

}
```

The first function, called `GenCorr`, is not discussed, because it is really technical and not really interesting here. The only thing to know is that it provides a proper correlation matrix: a positive-definite symmetrical matrix.

The function `flowrateModel` is loaded from the macro `UserFunctions.C` (the file can be found in `${URANIESYS}/`

```
gROOT->LoadMacro("UserFunctions.C");
```

Compared to the discussion in Section XIV.5.16.2, the instantiation of the attributes and the `TJohnsonRW` object is the same. The only difference is done when injecting the correlation matrix, which is done in these lines:

```
// Get a correlation matrix for the inputs
TMatrixD inCorr=GenCorr(8,true);
//Set the correlation
tjrw->setInputCorrelationMatrix(inCorr);
```

The computation of sensitivity indices can finally be done:

```
trbd->computeIndexes();
```

The rest is very common to all sensitivity macros discussed here: it will produce two plots (the first one being a histogram show below) and the console is also shown below for completness.

XIV.5.17.3 Graph

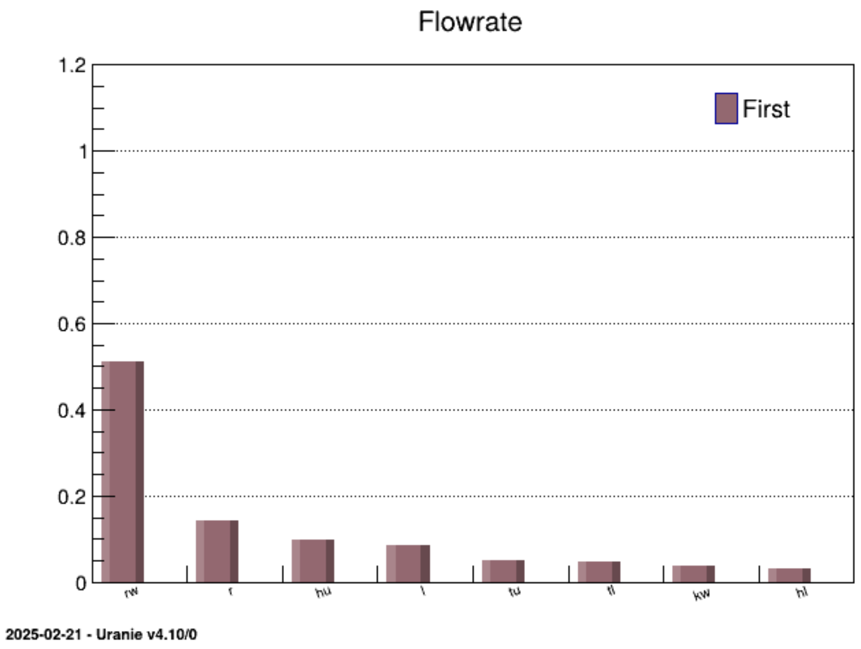


Figure XIV.59: Graph of the macro "sensitivityJohnsonRWCorrelatedFunctionFlowrate.C"

XIV.5.17.4 Console

```
Processing sensitivityJohnsonRWCorrelatedFunctionFlowrate.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025


8x8 matrix is as follows

     |      0    |      1    |      2    |      3    |      4    |
----------------------------------------------------------------------
   0 |          1      0.4488      0.2894    -0.05727      0.3627
   1 |     0.4488           1     0.02022     -0.2547      0.3764
   2 |     0.2894     0.02022           1      0.4982     -0.2078
   3 |   -0.05727     -0.2547      0.4982           1     -0.9199
   4 |     0.3627      0.3764     -0.2078     -0.9199           1
   5 |     0.2569     -0.1996      -0.288      0.1004     -0.1362
   6 |    -0.2167     -0.4425    0.004268      0.1412      -0.111
   7 |    -0.2948      0.1982   -0.001661      0.1943     -0.3131


     |      5    |      6    |      7    |
----------------------------------------------------------------------
   0 |     0.2569     -0.2167     -0.2948
   1 |    -0.1996     -0.4425      0.1982
   2 |     -0.288    0.004268   -0.001661
   3 |     0.1004      0.1412      0.1943
   4 |    -0.1362      -0.111     -0.3131
   5 |           1     -0.1943      0.2848
   6 |     -0.1943           1     -0.4415
   7 |      0.2848     -0.4415           1

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8530]
 <URANIE::WARNING> TDataServer::getTuple Error : There is no tree!
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${SOURCEDIR}/meTIER/sampler/souRCE/TSamplerStochastic.cxx] Line ↩
    [66]
 <URANIE::INFO> TSamplerStochastic::init: the TDS [tdsflowrate] contains data: we need to  ↩
    empty it !
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>
************************************************************
*    Row   *       Out *       Inp *    Method *     Value *
************************************************************
*        0 * flowrateM *        rw * JohnsonRW * 0.5113911 *
*        2 * flowrateM *         r * JohnsonRW * 0.1409327 *
*        4 * flowrateM *        tu * JohnsonRW * 0.0505176 *
*        6 * flowrateM *        tl * JohnsonRW * 0.0482684 *
*        8 * flowrateM *        hu * JohnsonRW * 0.0969452 *
*       10 * flowrateM *        hl * JohnsonRW * 0.0306674 *
*       12 * flowrateM *         l * JohnsonRW * 0.0836757 *
*       14 * flowrateM *        kw * JohnsonRW * 0.0376016 *
*       16 * flowrateM *    __R2__ * JohnsonRW * 0.9500813 *
*       18 * flowrateM *   __R2A__ * JohnsonRW * 0.9496787 *
************************************************************
==> 10 selected entries
```

XIV.5.18 Macro "sensitivityJohnsonRWJustCorrelationFakeFlowrate.C"

XIV.5.18.1 Objective

The objective of this macro is to perform a sensitivity analysis using the Johnson's relative weight method on a set of eight parameters which IS NOT the usual `flowrateModel` model. Indeed, compared to version detailled in Section XIV.5.17, the idea is here to correlate the input variables with a random correlation matrix and to translate this into a full correlation matrix, meaning defining a "fake" output by computing their covariance with every input as if this output was a perfect linear combination of these inputs.

An important particularity of this study is that no data are generated at all, it only uses the correlation matrix.

XIV.5.18.2 Macro Uranie

```
// Function defined to generate randomly a good, highly-correlated, correlation matrix for  ↩
    inputs
// and defibe the proper covariance with the output to do as if this output if a perfect  ↩
    linear combination
// of the inputs
TMatrixD GenCorr(int _nX=8, bool correlated=true)
{

    gRandom->SetSeed(__rdtsc());
    //Define a randomly filled matrix
    TMatrixD A(_nX,_nX);
    for (int i=0; i<_nX; i++)
        for (int j=0; j<_nX; j++)
            A(i,j)=gRandom->Gaus(0,1);

    // Compute AA^T and normalise it to get "covariance matrix"
    TMatrixD Gamma(A,TMatrixD::kMultTranspose,A);
    Gamma*=1./_nX;

    // Get the inverse of the diagonal matrix to do as if this was 1/sqrt(variance)
    TMatrixD Sig(_nX,_nX);
    for(int i=0;i<_nX;i++) Sig(i,i)=1./sqrt( Gamma(i,i) );

    // Compute the input correlation matrix
    TMatrixD inCorr(Sig*Gamma,TMatrixD::kMult, Sig);
    if( !correlated )
        inCorr.UnitMatrix();
    double varY=inCorr.Sum();

    // Proper correlation, output included
    TMatrixD Corr(_nX+1,_nX+1);
    Corr.UnitMatrix();
    // putting already defined input
    Corr.SetSub(0,0,inCorr);
    // Adjust the covariance of the output wrt to all inputs
    for(unsigned int i=0; i<_nX; i++)
    {
        double value=0;
        for(unsigned int j=0; j<_nX; j++)
            value+=inCorr(i,j);
        Corr(_nX,i) = Corr(i,_nX) = value / sqrt(varY);
    }

    return Corr;
}


void sensitivityJohnsonRWJustCorrelationFakeFlowrate()
{
  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsflowrate", "DataBase flowrate");
  tds->addAttribute("rw");
  tds->addAttribute("r");
  tds->addAttribute("tu");
  tds->addAttribute("tl");
  tds->addAttribute("hu");
  tds->addAttribute("hl");
  tds->addAttribute("l");
  tds->addAttribute("kw");
  // outputs
  tds->addAttribute("flowrateModel");
  tds->getAttribute("flowrateModel")->setOutput();


  // Get the full correlation matrix
  TMatrixD inCorr=GenCorr(8,true);

  // Johnson definition
  TJohnsonRW * tjrw = new TJohnsonRW(tds, "rw:r:tu:tl:hu:hl:l:kw", "flowrateModel");
  //Putting the newly defined correlation that states our output as a perfect linear ↩
      combination of inputs
  tjrw->setCorrelationMatrix(inCorr);
  tjrw->computeIndexes();

  // Get the results on screen
  tjrw->getResultTuple()->Scan("Out:Inp:Method:Value","Order==\"First\"");

  // Get the results as plots
  TCanvas *cc = new TCanvas("canhist", "histgramme");
  tjrw->drawIndexes("Flowrate", "", "nonewcanv,hist,first");
  cc->Print("appliUranieFakeFlowrateJohnsonRWCorrelationHistogram.png");

  TCanvas *ccc = new TCanvas("canpie", "TPie");
  tjrw->drawIndexes("Flowrate", "", "nonewcanv,pie");
  ccc->Print("appliUranieFakeFlowrateJohnsonRWCorrelationPie.png");

}
```

The first function, called `GenCorr`, is not discussed, because it is really technical and not really interesting here. The only thing to know is that it provides a proper correlation matrix: a positive-definite symmetrical matrix for the input and it computes the covariance for a "fake" output that would be a perfect linear combination of all the inputs.

Because of this, the function `flowrateModel` is not loaded anymore and the definition of the attributes is not the same: it is not necessary to use `TStochasticAttribute` because no data are generated here:

```
// Define the DataServer
TDataServer *tds = new TDataServer("tdsflowrate", "DataBase flowrate");
tds->addAttribute("rw");
tds->addAttribute("r");
tds->addAttribute("tu");
tds->addAttribute("tl");
tds->addAttribute("hu");
tds->addAttribute("hl");
tds->addAttribute("l");
tds->addAttribute("kw");
// outputs
tds->addAttribute("flowrateModel");
tds->getAttribute("flowrateModel")->setOutput();
```

Compared to the discussion in Section XIV.5.17.2, the only differences are the instantiation of the `TJohnsonRW` object and the method code to provide the correlation matrix.

```
// Get the full correlation matrix
TMatrixD inCorr=GenCorr(8,true);
// Johnson definition
TJohnsonRW * tjrw = new TJohnsonRW(tds, "rw:r:tu:tl:hu:hl:l:kw", "flowrateModel");
//Putting the newly defined correlation that states our output as a perfect linear ↩
    combination of inputs
tjrw->setCorrelationMatrix(inCorr);
```

The method called to provide the correlation matrix is `setCorrelationMatrix` and it means that the user give a full correlation matrix (input and output variables), on the contrary to the one used in Section XIV.5.17.2 which is `setInputCorrelationMatrix`, which set only the input correlation matrix.

The computation of sensitivity indices can finally be done:

```
trbd->computeIndexes();
```

The rest is very common to all sensitivity macros discussed here: it will produce two plots (the first one being a histogram show below) and the console is also shown below for completness.

XIV.5.18.3 Graph

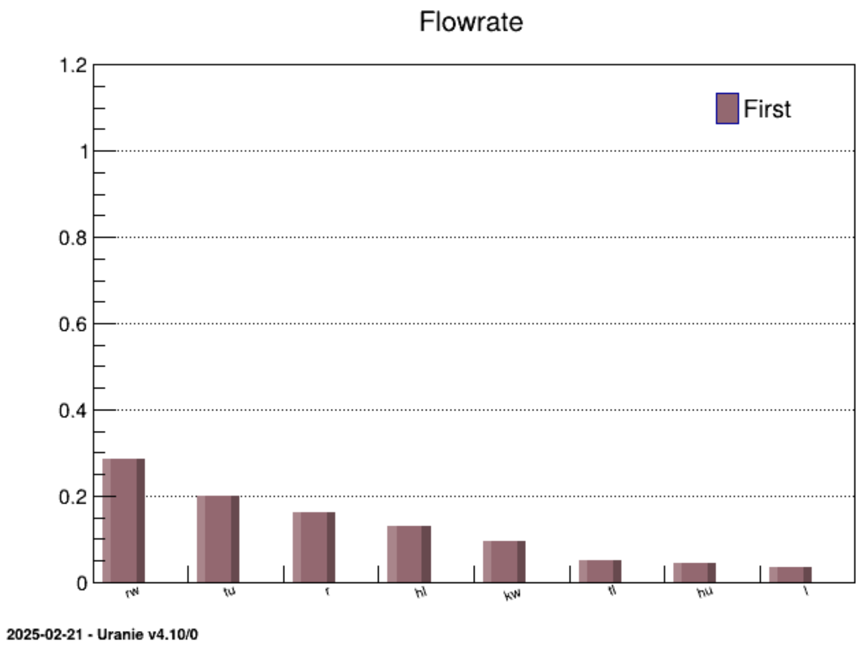


Figure XIV.60: Graph of the macro "sensitivityJohnsonRWJustCorrelationFakeFlowrate.C"

XIV.5.18.4 Console

```
Processing sensitivityJohnsonRWJustCorrelationFakeFlowrate.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[2201]
 <URANIE::WARNING> TDataServer::getNPatterns[tdsflowrate] WARNING
 <URANIE::WARNING>  The TTree is NULL
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/meTIER/sensitivity/souRCE/TSensitivity.cxx] Line ↩
    [165]
 <URANIE::WARNING> TSensitivity::constructor   ERROR The TTree is empty []
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8530]
 <URANIE::WARNING> TDataServer::getTuple Error : There is no tree!
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
************************************************************
*    Row   *       Out *       Inp *    Method *     Value *
```

```
************************************************************
*        0 * flowrateM *        rw * JohnsonRW * 0.2844573 *
*        2 * flowrateM *         r * JohnsonRW * 0.1600467 *
*        4 * flowrateM *        tu * JohnsonRW * 0.2004196 *
*        6 * flowrateM *        tl * JohnsonRW * 0.0512434 *
*        8 * flowrateM *        hu * JohnsonRW * 0.0446220 *
*       10 * flowrateM *        hl * JohnsonRW * 0.1295644 *
*       12 * flowrateM *         l * JohnsonRW * 0.0333889 *
*       14 * flowrateM *        kw * JohnsonRW * 0.0962574 *
*       16 * flowrateM *    __R2__ * JohnsonRW *         1 *
************************************************************
==> 9 selected entries
```

XIV.5.19 Macro "sensitivityHSICFunctionFlowrate.C"

XIV.5.19.1 Objective

The objective of this macro is to perform a sensitivity analysis using the HSIC method on a set of eight parameters used in the `flowrateModel` model described in Section IV.1.2.1.

XIV.5.19.2 Macro Uranie

```
void sensitivityHSICFunctionFlowrate(){

gROOT->LoadMacro("UserFunctions.C");

// Define the DataServer
TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

// Generation of the sample (it can be a given sample).
Int_t nS = 500;
TSampling *sampling = new TSampling(tds, "lhs", nS);
sampling->generateSample();

TLauncherFunction * tlf = new TLauncherFunction(tds, "flowrateModel");
tlf->setDrawProgressBar(kFALSE);
tlf->run();

// Create a THSIC object, compute indexes and print results
THSIC * thsic = new THSIC(tds, "rw:r:tu:tl:hu:hl:l:kw","flowrateModel");
thsic->computeIndexes("quiet");
thsic->getResultTuple()->SetScanField(60);
thsic->getResultTuple()->Scan("Out:Inp:Method:Order:Value:CILower:CIUpper");

// Print HSIC indexes
TCanvas  *can = new TCanvas("c1", "Graph for the Macro sensitivityHSICFunctionFlowrate" ↩
    ,5,64,1270,667);
thsic->drawIndexes("Flowrate", "", "hist,first,nonewcanv");

}
```

The function `flowrateModel` is loaded from the macro `UserFunctions.C` (the file can be found in `${URANIESYS}/share

```
gROOT->LoadMacro("UserFunctions.C");
```

Each parameter is related to the `TDataServer` as a `TAttribute` and obeys an uniform law on specific interval

```
// Define the DataServer
TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

HSIC does not need a specific DOE, it works with a given sample. We generate a sample with `TSampling` and we evaluate it using `TLauncherFunction`

```
Int_t nS = 500;
TSampling *sampling = new TSampling(tds, "lhs", nS);
sampling->generateSample();

TLauncherFunction * tlf = new TLauncherFunction(tds, "flowrateModel");
tlf->setDrawProgressBar(kFALSE);
tlf->run();
```

To instantiate the `THSIC` object, one uses the `TDataServer`,the name of the input and output variables:

```
THSIC * thsic = new THSIC(tds, "rw:r:tu:tl:hu:hl:l:kw","flowrateModel");
```

Computation of sensitivity indexes:

```
thsic->computeIndexes();
```

It will produce one plots containing the HSIC indexes and the p-value to test the Independance between inputs and outputs.The console is also shown below for completness.

XIV.5.19.3 Graph

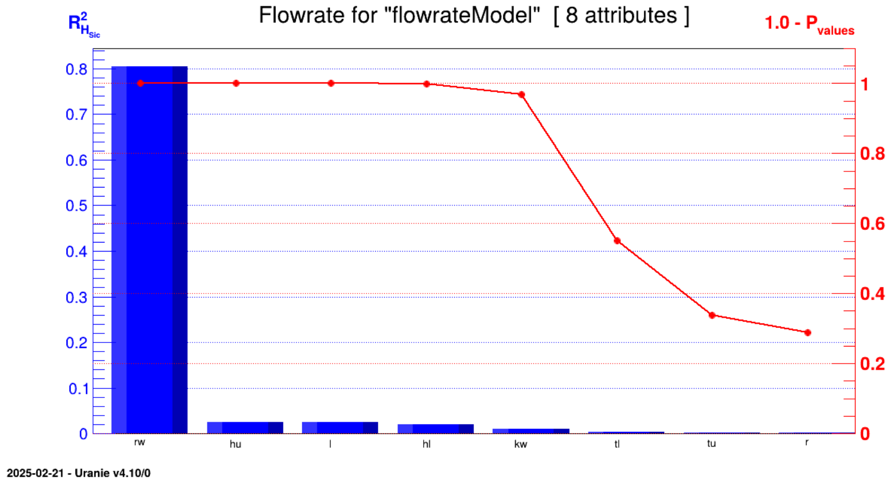


Figure XIV.61: Graph of the macro "sensitivityHSICFunctionFlowrate.C"

XIV.5.19.4 Console

```
Processing sensitivityHSICFunctionFlowrate.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

*************************************************************************************
*    Row   *    Out *       Inp * Metho *      Order *     Value *  CILower *  CIUpper *
*************************************************************************************
*        0 * flowra *        rw *  HSic *     R2HSic * 0.804431 *        -1 *        -1 *
*        1 * flowra *        rw *  HSic *       HSic * 0.072723 *        -1 *        -1 *
*        2 * flowra *        rw *  HSic *    pValues * 6.8e-106 *        -1 *        -1 *
*        3 * flowra *         r *  HSic *     R2HSic * 0.002238 *        -1 *        -1 *
*        4 * flowra *         r *  HSic *       HSic * 0.000202 *        -1 *        -1 *
*        5 * flowra *         r *  HSic *    pValues * 0.712174 *        -1 *        -1 *
*        6 * flowra *        tu *  HSic *     R2HSic * 0.002528 *        -1 *        -1 *
*        7 * flowra *        tu *  HSic *       HSic * 0.000228 *        -1 *        -1 *
*        8 * flowra *        tu *  HSic *    pValues * 0.662668 *        -1 *        -1 *
*        9 * flowra *        tl *  HSic *     R2HSic * 0.003806 *        -1 *        -1 *
*       10 * flowra *        tl *  HSic *       HSic * 0.000344 *        -1 *        -1 *
*       11 * flowra *        tl *  HSic *    pValues * 0.449068 *        -1 *        -1 *
*       12 * flowra *        hu *  HSic *     R2HSic * 0.025554 *        -1 *        -1 *
*       13 * flowra *        hu *  HSic *       HSic * 0.002309 *        -1 *        -1 *
*       14 * flowra *        hu *  HSic *    pValues * 3.13e-05 *        -1 *        -1 *
*       15 * flowra *        hl *  HSic *     R2HSic * 0.020243 *        -1 *        -1 *
*       16 * flowra *        hl *  HSic *       HSic * 0.001829 *        -1 *        -1 *
*       17 * flowra *        hl *  HSic *    pValues * 0.000445 *        -1 *        -1 *
*       18 * flowra *         l *  HSic *     R2HSic * 0.024934 *        -1 *        -1 *
*       19 * flowra *         l *  HSic *       HSic * 0.002253 *        -1 *        -1 *
*       20 * flowra *         l *  HSic *    pValues * 5.64e-05 *        -1 *        -1 *
*       21 * flowra *        kw *  HSic *     R2HSic * 0.010567 *        -1 *        -1 *
*       22 * flowra *        kw *  HSic *       HSic * 0.000955 *        -1 *        -1 *
*       23 * flowra *        kw *  HSic *    pValues * 0.032459 *        -1 *        -1 *
*************************************************************************************
```

XIV.5.20 Macro "sensitivitySobolRankFunctionFlowrate.C"

XIV.5.20.1 Objective

The objective of this macro is to perform a sensitivity analysis using the SobolRank method on a set of eight parameters used in the `flowrateModel` model described in Section IV.1.2.1.

XIV.5.20.2 Macro Uranie

```
void sensitivitySobolRankFunctionFlowrate(){

gROOT->LoadMacro("UserFunctions.C");

// Define the DataServer
TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

// Generation of the sample (it can be a given sample).
Int_t nS = 500;
TSampling *sampling = new TSampling(tds, "lhs", nS);
sampling->generateSample();

TLauncherFunction * tlf = new TLauncherFunction(tds, "flowrateModel");
tlf->setDrawProgressBar(kFALSE);
tlf->run();

// Create a TSobolRank object, compute indexes and print results
TSobolRank * tsobolrank = new TSobolRank(tds, "rw:r:tu:tl:hu:hl:l:kw","flowrateModel");
tsobolrank->computeIndexes("quiet");
tsobolrank->getResultTuple()->SetScanField(60);
tsobolrank->getResultTuple()->Scan("Out:Inp:Method:Order:Value:CILower:CIUpper");

// Print Sobol indexes
TCanvas  *can = new TCanvas("c1", "Graph for the Macro sensitivitySobolRankFunctionFlowrate ↩
    ",5,64,1270,667);
tsobolrank->drawIndexes("Flowrate", "", "hist,first,nonewcanv");

}
```

The function `flowrateModel` is loaded from the macro `UserFunctions.C` (the file can be found in `${URANIESYS}/share`

```
gROOT->LoadMacro("UserFunctions.C");
```

Each parameter is related to the `TDataServer` as a `TAttribute` and obeys an uniform law on specific interval

```
// Define the DataServer
TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

SobolRank does not need a specific DOE, it works with a given sample. We generate a sample with `TSampling` and we evaluate it using `TLauncherFunction`

```
Int_t nS = 500;
TSampling *sampling = new TSampling(tds, "lhs", nS);
sampling->generateSample();

TLauncherFunction * tlf = new TLauncherFunction(tds, "flowrateModel");
tlf->setDrawProgressBar(kFALSE);
tlf->run();
```

To instantiate the `TSobolRank` object, one uses the `TDataServer`,the name of the input and output variables:

```
TSobolRank * tsobolrank = new TSobolRank(tds, "rw:r:tu:tl:hu:hl:l:kw","flowrateModel");
```

Computation of sensitivity indexes:

```
tsobolrank->computeIndexes();
```

It will produce one plots containing the HSIC indexes and the p-value to test the Independance between inputs and outputs.The console is also shown below for completness.

XIV.5.20.3 Graph

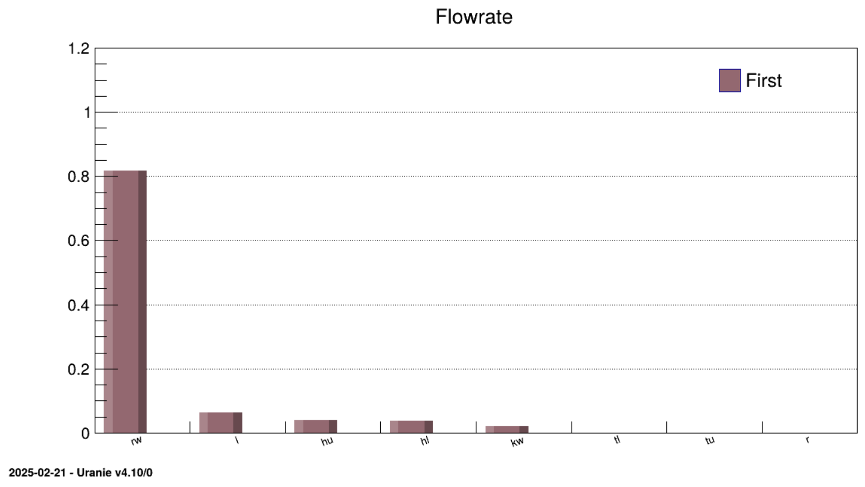


Figure XIV.62: Graph of the macro "sensitivitySobolRankFunctionFlowrate.C"

XIV.5.20.4 Console

```
Processing sensitivitySobolRankFunctionFlowrate.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

************************************************************************************
*    Row   *    Out *      Inp * Metho *     Order *    Value *  CILower *  CIUpper *
************************************************************************************
*        0 * flowra *       rw * Sobol *     First * 0.816872 *       -1 *       -1 *
*        1 * flowra *        r * Sobol *     First * -0.04759 *       -1 *       -1 *
*        2 * flowra *       tu * Sobol *     First * -0.04164 *       -1 *       -1 *
*        3 * flowra *       tl * Sobol *     First * -0.00414 *       -1 *       -1 *
*        4 * flowra *       hu * Sobol *     First * 0.041272 *       -1 *       -1 *
*        5 * flowra *       hl * Sobol *     First * 0.036748 *       -1 *       -1 *
*        6 * flowra *        l * Sobol *     First * 0.062867 *       -1 *       -1 *
*        7 * flowra *       kw * Sobol *     First * 0.021273 *       -1 *       -1 *
************************************************************************************
```

XIV.5.21 Macro "sensitivityCramerVonMisesFunctionFlowrate.C"

XIV.5.21.1 Objective

The objective of this macro is to perform a sensitivity analysis using the Cramer Von Mises method on a set of eight parameters used in the `flowrateModel` model described in Section IV.1.2.1.

XIV.5.21.2 Macro Uranie

```
{

  gROOT->LoadMacro("UserFunctions.C");

  // Define the DataServer
  TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  Int_t ns = 10000;
  TCramerVonMises * tcvm = new TCramerVonMises(tds, "flowrateModel", ns, "rw:r:tu:tl:hu:hl: ↩
      l:kw", "flowrateModel", "pouet");
  tcvm->setDrawProgressBar(kFALSE);
  tcvm->computeIndexes();

  tcvm->getResultTuple()->Scan("*","Method==\"CramerVonMises\"");

  TCanvas *cc = new TCanvas("c1", "histgramme",5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2,1);
  pad->cd(1);
  tcvm->drawIndexes("Flowrate", "", "nonewcanv,hist,all");

  pad->cd(2);
  tcvm->drawIndexes("Flowrate", "", "nonewcanv,pie,first");

    }
```

The function `flowrateModel` is loaded from the macro `UserFunctions.C` (the file can be found in `${URANIESYS}/share`

```
gROOT->LoadMacro("UserFunctions.C");
```

Each parameter is related to the `TDataServer` as a `TAttribute` and obeys an uniform law on specific interval

```
// Define the DataServer
TDataServer *tds = new TDataServer("tdsflowreate", "DataBase flowreate");
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

CramerVonMises is a extension of the pick and freeze DOE use for Sobol method. To instantiate the `TCramerVonMises` object, one uses the `TDataServer`, the name of the function and the number of samplings needed to perform sensitivity analysis (here ns=10000):

```
TCramerVonMises * tcvm = new TCramerVonMises(tds, "flowrateModel", ns, "rw:r:tu:tl:hu:hl:l: ↩
    kw", "flowrateModel");
```

Computation of sensitivity indexes:

```
tcvm->computeIndexes();
```

As CramerVonMises is a extension of the pick and freeze DOE use for Sobol method, the Sobol indices are also computed.

XIV.5.21.3 Graph

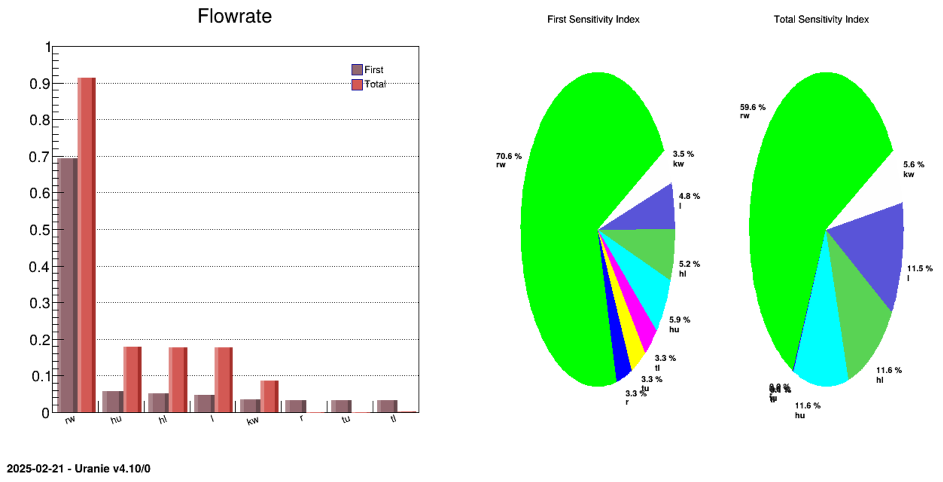


Figure XIV.63: Graph of the macro "sensitivityCramerVonMisesFunctionFlowrate.C"

XIV.5.21.4 Console

```
Processing sensitivityCramerVonMisesFunctionFlowrate.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8530]
 <URANIE::WARNING> TDataServer::getTuple Error : There is no tree!
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${SOURCEDIR}/meTIER/sampler/souRCE/TSamplerStochastic.cxx] Line ↩
    [66]
 <URANIE::INFO> TSamplerStochastic::init: the TDS [tdsflowreate] contains data: we need to ↩
    empty it !
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>
 ** Case of Output atty [flowrateModel]  nSimPerIndex 910
 ** Input att [rw] First CVM [0.694231] Total Order CVM[0.913107]
 ** Input att [r] First CVM [0.0327274] Total Order CVM[0.000562551]
 ** Input att [tu] First CVM [0.0326926] Total Order CVM[0]
 ** Input att [tl] First CVM [0.0326247] Total Order CVM[0.00153446]
 ** Input att [hu] First CVM [0.0582536] Total Order CVM[0.17845]
 ** Input att [hl] First CVM [0.0515417] Total Order CVM[0.177485]
 ** Input att [l] First CVM [0.0469543] Total Order CVM[0.176094]
 ** Input att [kw] First CVM [0.0343777] Total Order CVM[0.0857252]
*********************************************************************************************************

*    Row   *   Out.Out *   Inp.Inp * Order.Ord * Method.Me * Algo.Algo * Value.Val *  ↩
   CILower.C * CIUpper.C *
*********************************************************************************************************

*       66 * flowrateM *        rw *  FirstCVM * CramerVon * --first-- * 0.6942309 *  ↩
          -1 *        -1 *
*       67 * flowrateM *        rw *  TotalCVM * CramerVon * --total-- * 0.9131074 *  ↩
          -1 *        -1 *
*       68 * flowrateM *         r *  FirstCVM * CramerVon * --first-- * 0.0327273 *  ↩
          -1 *        -1 *
*       69 * flowrateM *         r *  TotalCVM * CramerVon * --total-- * 0.0005625 *  ↩
          -1 *        -1 *
*       70 * flowrateM *        tu *  FirstCVM * CramerVon * --first-- * 0.0326926 *  ↩
          -1 *        -1 *
*       71 * flowrateM *        tu *  TotalCVM * CramerVon * --total-- *         0 *  ↩
          -1 *        -1 *
*       72 * flowrateM *        tl *  FirstCVM * CramerVon * --first-- * 0.0326246 *  ↩
          -1 *        -1 *
*       73 * flowrateM *        tl *  TotalCVM * CramerVon * --total-- * 0.0015344 *  ↩
          -1 *        -1 *
*       74 * flowrateM *        hu *  FirstCVM * CramerVon * --first-- * 0.0582536 *  ↩
          -1 *        -1 *
*       75 * flowrateM *        hu *  TotalCVM * CramerVon * --total-- * 0.1784495 *  ↩
          -1 *        -1 *
*       76 * flowrateM *        hl *  FirstCVM * CramerVon * --first-- * 0.0515417 *  ↩
          -1 *        -1 *
*       77 * flowrateM *        hl *  TotalCVM * CramerVon * --total-- * 0.1774852 *  ↩
          -1 *        -1 *
*       78 * flowrateM *         l *  FirstCVM * CramerVon * --first-- * 0.0469543 *  ↩
          -1 *        -1 *
*       79 * flowrateM *         l *  TotalCVM * CramerVon * --total-- * 0.1760940 *  ↩
          -1 *        -1 *
*       80 * flowrateM *        kw *  FirstCVM * CramerVon * --first-- * 0.0343777 *  ↩
          -1 *        -1 *
*       81 * flowrateM *        kw *  TotalCVM * CramerVon * --total-- * 0.0857252 *  ↩
          -1 *        -1 *
*       82 * flowrateM *   __sum__ *  FirstCVM * CramerVon * --first-- * 0.9834030 *  ↩
          -1 *        -1 *
*       83 * flowrateM *   __sum__ *  TotalCVM * CramerVon * --total-- * 1.5329584 *  ↩
          -1 *        -1 *
*********************************************************************************************************

==> 18 selected entries
```

XIV.6 Macros Modeler

XIV.6.1 Macro "`modelerCornellLinearRegression.C`"

XIV.6.1.1 Objective

The objective of the macro is to build a multilinear regression between the predictors related to the normalisation of the variables **x1**, **x2**, **x3**, **x4**, **x5**, **x6**, **x7** and a target variable **y** from the database contained in the ASCII file `cornell.dat`:

```
#NAME: cornell
#TITLE: Dataset Cornell 1990
#COLUMN_NAMES: x1 | x2 | x3 | x4 | x5 | x6 | x7 | y
#COLUMN_TITLES: x_{1} | x_{2} | x_{3} | x_{4} | x_{5} | x_{6} | x_{7} | y

0.00 0.23 0.00 0.00 0.00 0.74 0.03 98.7
0.00 0.10 0.00 0.00 0.12 0.74 0.04 97.8
0.00 0.00 0.00 0.10 0.12 0.74 0.04 96.6
0.00 0.49 0.00 0.00 0.12 0.37 0.02 92.0
0.00 0.00 0.00 0.62 0.12 0.18 0.08 86.6
0.00 0.62 0.00 0.00 0.00 0.37 0.01 91.2
0.17 0.27 0.10 0.38 0.00 0.00 0.08 81.9
0.17 0.19 0.10 0.38 0.02 0.06 0.08 83.1
0.17 0.21 0.10 0.38 0.00 0.06 0.08 82.4
0.17 0.15 0.10 0.38 0.02 0.10 0.08 83.2
0.21 0.36 0.12 0.25 0.00 0.00 0.06 81.4
0.00 0.00 0.00 0.55 0.00 0.37 0.08 88.1
```

XIV.6.1.2 Macro Uranie

```
{
  TDataServer * tds = new TDataServer();
  tds->fileDataRead("cornell.dat");

  tds->normalize("*", "cr");  // ("x1:x2:x3:x4:x5:x6:y", "cr", TDataServer::kCR)

  TCanvas *c = new TCanvas("c1", "Graph for the Macro modeler",5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2);
  pad->cd(1);
  tds->draw("y:x7");

  TLinearRegression *tlin = new TLinearRegression(tds,"x1cr:x2cr:x3cr:x4cr:x5cr:x6cr", "ycr ↩
      ", "nointercept");
  tlin->estimate();

 pad->cd(2);
 tds->draw("ycr:ycrhat");

}
```

The ASCII data file `cornell.dat` is loaded in the `TDataServer`:

```
TDataServer * tds = new TDataServer();
tds->fileDataRead("cornell.dat");
```

The variables are all normalised with a standardised method. The normalised attributes are created with the *cr* extension:

```
tds->normalize("*", "cr");
```

The linear regression is initialised and characteristic values are computed for each normalised variable with the `estimate` method. The regression is built with no intercept:

```
TLinearRegression *tlin = new TLinearRegression(tds,"x1cr:x2cr:x3cr:x4cr:x5cr:x6cr", "ycr", ↩
     "nointercept");
tlin->estimate();
```

XIV.6.1.3 Graph

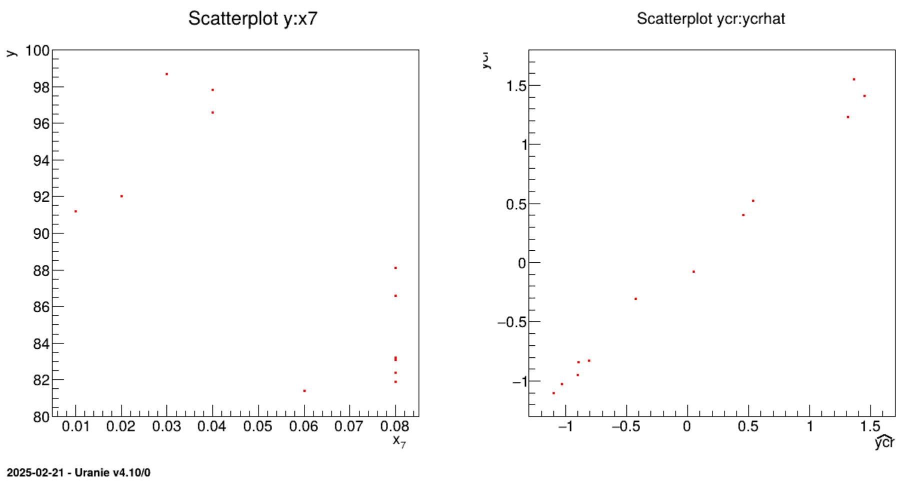


Figure XIV.64: Graph of the macro "`modelerCornellLinearRegression.C`"

XIV.6.2 Macro "**`modelerFlowrateLinearRegression.C`**"

XIV.6.2.1 Objective

The objective of this macro is to build a linear regression between a predictor **rw** and a target variable **yhat** from the database contained in the ASCII file `flowrate_sampler_launcher_500.dat` defining values for the eight variables described in Section IV.1.2.1 on 500 patterns.

XIV.6.2.2 Macro Uranie

```
{
  TDataServer * tds = new TDataServer();
  tds->fileDataRead("flowrate_sampler_launcher_500.dat");

  TCanvas *c = new TCanvas("c1", "Graph for the Macro modeler",5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2);
  pad->cd(1); tds->draw("yhat:rw");

  //   tds->getAttribute("yhat")->setOutput();

  TLinearRegression *tlin = new TLinearRegression(tds, "rw","yhat");
  tlin->estimate();

  pad->cd(2); tds->draw("yhathat:yhat");

  tds->startViewer();

}
```

The `TDataServer` is loaded with the database contained in the file `flowrate_sampler_launcher_500.dat` with the `fileDataRead` method:

```
TDataServer * tds = new TDataServer();
tds->fileDataRead("flowrate_sampler_launcher_500.dat");
```

The linear regression is initialised and characteristic values are computed for **rw** with the `estimate` method:

```
TLinearRegression *tlin = new TLinearRegression(tds, "rw","yhat");
tlin->estimate();
```

XIV.6.2.3 Graph

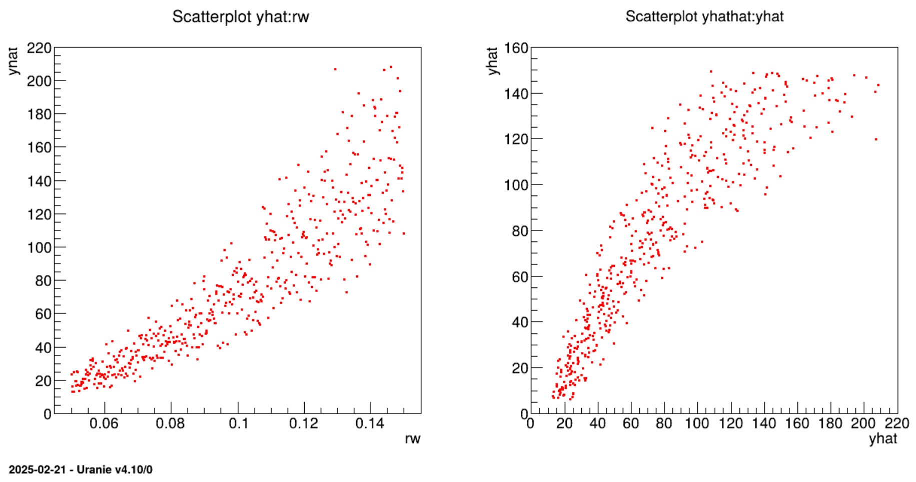


Figure XIV.65: Graph of the macro "`modelerFlowrateLinearRegression.C`"

XIV.6.3 Macro "modelerFlowrateMultiLinearRegression.C"

XIV.6.3.1 Objective

The objective of the macro is to build a multilinear regression between the predictors r_ω, r, T_u, T_l, H_u, H_l, L, K_ω and a target variable **yhat** from the database contained in the ASCII file `flowrate_sampler_launcher_500.dat` defining values for these eight variables described in Section IV.1.2.1 on 500 patterns. Parameters of the regression are then exported in a .C file `_FileContainingFunction_.C`:

```
void MultiLinearFunction(double *param, double *res)
{
  //////////////////////////////
  //
  //     *********************************************
  //     ** Uranie v3.12/0 -  Date : Thu Jan 04, 2018
  //     ** Export Modeler : Modeler[LinearRegression]Tds[tdsFlowrate]Predictor[rw:r:tu:tl: ↩
      hu:hl:l:kw]Target[yhat]
  //     ** Date : Tue Jan  9 12:08:27 2018

  //     *********************************************
  //
  //     *********************************************
  //     ** TDataServer
  //     **          Name   : tdsFlowrate
  //     **          Title : Design of experiments for Flowrate
  //      **      Patterns : 500
  //      **    Attributes : 10
  //      *********************************************
  //
  //      INPUT   : 8
  //              rw : Min[0.050069828419999997] Max[0.14991758599999999] Unit[]
  //              r : Min[147.90551809999999] Max[49906.309529999999] Unit[]
  //              tu : Min[63163.702980000002] Max[115568.17200000001] Unit[]
  //              tl : Min[63.169232649999998] Max[115.90147020000001] Unit[]
  //              hu : Min[990.00977509999996] Max[1109.786159] Unit[]
  //              hl : Min[700.14498509999999] Max[819.81105760000003] Unit[]
  //              l : Min[1120.3428429999999] Max[1679.3424649999999] Unit[]
  //              kw : Min[9857.3689890000005] Max[12044.00546] Unit[]
  //
  //      OUTPUT : 1
  //              yhat : Min[13.09821] Max[208.25110000000001] Unit[]
  //
  //////////////////////////////
  //
  //      QUALITY   :
  //
  //               R2   = 0.948985
  //               R2a  = 0.948154
  //               Q2   = 0.946835
  //
  //////////////////////////////

  // Intercept
  double y = -156.03;

  // Attribute : rw
  y += param[0]*1422.21;

  // Attribute : r
  y += param[1]*-3.07412e-07;

  // Attribute : tu
  y += param[2]*2.15208e-06;

  // Attribute : tl
  y += param[3]*-0.00498512;

  // Attribute : hu
  y += param[4]*0.261104;

  // Attribute : hl
  y += param[5]*-0.254419;

  // Attribute : l
  y += param[6]*-0.0557145;

  // Attribute : kw
  y += param[7]*0.00813552;

  // Return the value
  res[0] = y;
}
```

This file contains a `MultiLinearFunction` function. Comparison is made with a database generated from random variables obeying uniform laws and output variable calculated with this database and the `MultiLinearFunction` function.

XIV.6.3.2 Macro Uranie

```
{
  TDataServer * tds = new TDataServer();
  tds->fileDataRead("flowrate_sampler_launcher_500.dat");

  TCanvas *c = new TCanvas("c1", "Graph for the Macro modeler",5,64,1270,667);
  c->Divide(2);
  c->cd(1); tds->draw("yhat:rw");

  // tds->getAttribute("yhat")->setOutput();

  TLinearRegression *tlin = new TLinearRegression(tds, "rw:r:tu:tl:hu:hl:l:kw", "yhat");
  tlin->estimate();

  tlin->exportFunction("c++", "_FileContainingFunction_", "MultiLinearFunction");

  // Define the DataServer
  TDataServer *tds2 = new TDataServer("tdsFlowrate", "Design of experiments for Flowrate");

  // Add the study attributes
  tds2->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds2->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
  tds2->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds2->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds2->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds2->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds2->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds2->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  Int_t nS = 1000;
  //   \test Générateur de plan d'expérience
  TSampling *sampling = new TSampling(tds2, "lhs", nS);
  sampling->generateSample();

  gROOT->LoadMacro("_FileContainingFunction_.C");

  // Create a TLauncherFunction from a TDataServer and an analytical function
  // Rename the outpout attribute "ymod"
  TLauncherFunction * tlf = new TLauncherFunction(tds2, "MultiLinearFunction","","ymod");
  // Evaluate the function on all the design of experiments
  tlf->run();

  c->Clear();
  tds2->getTuple()->SetMarkerColor(kBlue);
  tds2->getTuple()->SetMarkerStyle(8);
  tds2->getTuple()->SetMarkerSize(1.2);

  tds->getTuple()->SetMarkerColor(kGreen);
  tds->getTuple()->SetMarkerStyle(8);
  tds->getTuple()->SetMarkerSize(1.2);

  tds2->draw("ymod:rw");
  tds->draw("yhat:rw","","same");

}
```

The `TDataServer` is loaded with the database contained in the file `flowrate_sampler_launcher_500.dat` with the `fileDataRead` method:

```
TDataServer * tds = new TDataServer();
```

```
tds->fileDataRead("flowrate_sampler_launcher_500.dat");
```

The linear regression is initialised and characteristic values are computed for each variable with the `estimate` method:

```
TLinearRegression *tlin = new TLinearRegression(tds, "rw:r:tu:tl:hu:hl:l:kw", "yhat");
tlin->estimate();
```

The model is exported in an external file in C++ language `_FileContainingFunction_.C` where the function name is `MultiLinearFunction`:

```
tlin->exportFunction("c++", "_FileContainingFunction_", "MultiLinearFunction");
```

A second `TDataServer` is created. The previous variables then obey uniform laws and are linked as `TAttribute` in this new `TDataServer`:

```
tds2->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds2->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
tds2->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
tds2->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
tds2->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
tds2->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
tds2->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
tds2->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));
```

A sampling is realised with a LHS method on 1000 patterns:

```
TSampling *sampling = new TSampling(tds2, "lhs", nS=1000);
sampling->generateSample();
```

The previously exported macro `_FileContainingFunction_.C` is loaded so as to perform calculation over the database in the second `TDataServer` with the function `MultiLinearFunction`. Results are stored in the **ymod** variable:

```
gROOT->LoadMacro("_FileContainingFunction_.C");
TLauncherFunction * tlf = new TLauncherFunction(tds2, "MultiLinearFunction", "", "ymod");
tlf->run();
```

XIV.6.3.3 Graph

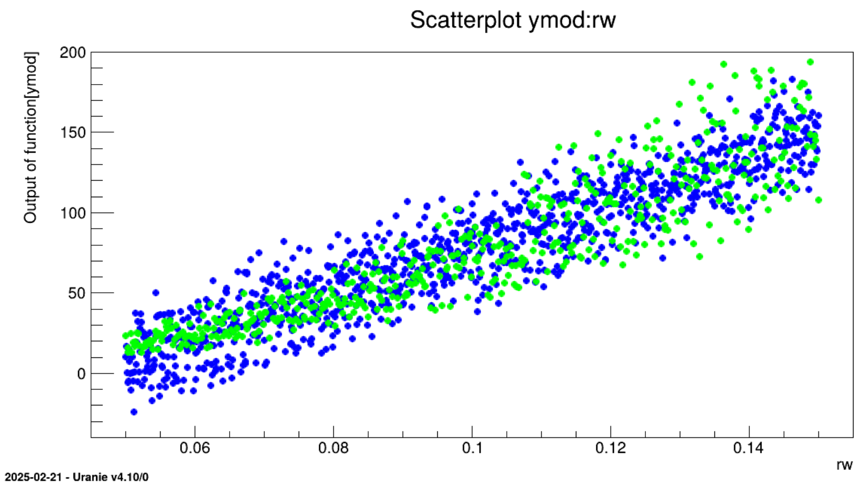


Figure XIV.66: Graph of the macro "modelerFlowrateMultiLinearRegression.C"

XIV.6.4 Macro "**modelerFlowrateNeuralNetworks.C**"

XIV.6.4.1 Objective

The objective of this macro is to build a surrogate model (an Artificial Neural Network) from a database. From a first database created from an ASCII file `flowrate_sampler_launcher_500.dat` (which defines values for eight variables described in Section IV.1.2.1 on 500 patterns), two ASCII files are created: one with the 300st patterns (`_flowrate_sampler_launcher_app_.dat`), the other with the 200 last patterns (`_flowrate_sampler_launcher_val_.dat`). The surrogate model is built with the database extracted from the first of the two files, the second allowing to perform calculations with a function.

XIV.6.4.2 Macro Uranie

```
{
  // Create a TDataServer
  // Load a database in an ASCII file
  TDataServer * tdsu = new TDataServer("tdsu","tds u");
  tdsu->fileDataRead("flowrate_sampler_launcher_500.dat");

  //Create database
  tdsu->exportData("_flowrate_sampler_launcher_app_.dat", "","tdsFlowrate__n__iter__<=300") ↩
      ;
  tdsu->exportData("_flowrate_sampler_launcher_val_.dat", "","tdsFlowrate__n__iter__>300");

  TDataServer * tds = new TDataServer("tdsFlowrate", "tds for flowrate");
  tds->fileDataRead("_flowrate_sampler_launcher_app_.dat");

  TCanvas *c = new TCanvas("c1", "Graph for the Macro modeler",5,64,1270,667);
  // Buils a surrogate model (Artificial Neural Networks) from the DataBase
  TANNModeler* tann = new TANNModeler(tds, "rw:r:tu:tl:hu:hl:l:kw,3,yhat");
  tann->setFcnTol(1e-5);
  //tann->setLog();
  tann->train(3, 2, "test");
  tann->exportFunction("pmmlc++","uranie_ann_flowrate","ANNflowrate");

  gROOT->LoadMacro("uranie_ann_flowrate.C");

  TDataServer * tdsv = new TDataServer();
  tdsv->fileDataRead("_flowrate_sampler_launcher_val_.dat");

  cout << tdsv->getNPatterns() << endl;

  // evaluate the surrogate model on the database
  TLauncherFunction * tlf = new TLauncherFunction(tdsv, "ANNflowrate", "rw:r:tu:tl:hu:hl:l: ↩
      kw", "yann");
  tlf->run();

  tdsv->startViewer();

  tdsv->Draw("yann:yhat");
  //  tdsv->draw("yhat");

}
```

The main TDataServer loads the main ASCII data file flowrate_sampler_launcher_500.dat

```
TDataServer * tdsu = new TDataServer("tdsu","tds u");
tdsu->fileDataRead("flowrate_sampler_launcher_500.dat");
```

The database is split in two parts by exporting the 300st patterns in a file and the remaining 200 in another one:

```
tdsu->exportData("_flowrate_sampler_launcher_app_.dat", "","tdsFlowrate__n__iter__<=300");
tdsu->exportData("_flowrate_sampler_launcher_val_.dat", "","tdsFlowrate__n__iter__>300");
```

A second TDataServer loads _flowrate_sampler_launcher_app_.dat and builds the surrogate model over all the variables:

```
TDataServer * tds = new TDataServer("tds", "tds for flowrate");
tds->fileDataRead("_flowrate_sampler_launcher_app_.dat");
TANNModeler* tann = new TANNModeler(tds, "rw:r:tu:tl:hu:hl:l:kw,3,yhat");
tann->setFcnTol(1e-5);
tann->train(3, 2, "test");
```

The model is exported in an external file in C++ language "uranie_ann_flowrate.C where the function name is ANNflowrate:

```
tann->exportFunction("c++", "uranie_ann_flowrate.C","ANNflowrate");
```

The model is loaded from the macro "uranie_ann_flowrate.C and applied on the second database with the function ANNflowrate:

```
gROOT->LoadMacro("uranie_ann_flowrate.C");
TDataServer * tdsv = new TDataServer();
tdsv->fileDataRead("_flowrate_sampler_launcher_val_.dat");
TLauncherFunction * tlf = new TLauncherFunction(tdsv, "ANNflowrate", "rw:r:tu:tl:hu:hl:l:kw ↩
    ", "yann");
tlf->run();
```

XIV.6.4.3 Graph

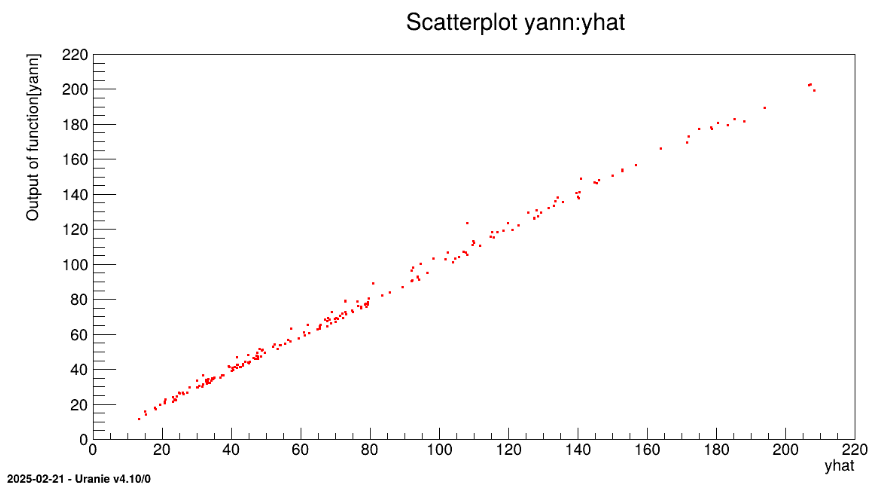


Figure XIV.67: Graph of the macro "modelerFlowrateNeuralNetworks.C"

XIV.6.4.4 Console

```
Processing modelerFlowrateNeuralNetworks.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[759]
 <URANIE::WARNING> TDataServer::fileDataRead: Expected iterator tdsu__n__iter__ not found ↩
    but tdsFlowrate__n__iter__ looks like an URANIE iterator => Will be used as so.
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 ** TANNModeler::train niter[3] ninit[2]
 ** init the ANN
 ** Input  (1) Name[rw] Min[0.101018] Max[0.0295279]
 ** Input  (2) Name[r] Min[25668.6] Max[14838.3]
 ** Input  (3) Name[tu] Min[89914.4] Max[14909]
 ** Input  (4) Name[tl] Min[89.0477] Max[15.0121]
 ** Input  (5) Name[hu] Min[1048.92] Max[34.8615]
 ** Input  (6) Name[hl] Min[763.058] Max[33.615]
 ** Input  (7) Name[l] Min[1401.09] Max[163.611]
 ** Input  (8) Name[kw] Min[10950.2] Max[635.503]
 ** Output  (1) Name[yhat] Min[78.0931] Max[44.881]
 ** Tolerance  (1e-05)
 ** sHidden    (3) (3)
 ** Nb Weights (31)
 ** ActivationFunction[LOGISTIC]
 **

 ** iter[1/3] : *= : mse_min[0.00150425]
 ** iter[2/3] : ** : mse_min[0.0024702]
 ** iter[3/3] : *= : mse_min[0.00122167]
 ** solutions : 3
 ** isol[1] iter[0] learn[0.00126206] test[0.00150425] *
 ** isol[2] iter[1] learn[0.00152949] test[0.0024702]
 ** isol[3] iter[2] learn[0.00107926] test[0.00122167] *
 ** CPU training finished. Total elapsed time: 1.49 sec

*******************************
*** TModeler::exportFunction lang[pmmlc++] file[uranie_ann_flowrate] name[ANNflowrate]  ↩
    soption[]

*******************************

*******************************
*** exportFunction lang[pmmlc++] file[uranie_ann_flowrate] name[ANNflowrate]

*******************************
PMML Constructor: uranie_ann_flowrate.pmml
*** End Of exportModelPMML
*******************************
*** End Of exportFunction
*******************************
```

XIV.6.5 Macro "modelerFlowrateNeuralNetworksLoadingPMML.C"

XIV.6.5.1 Objective

The objective of this macro is to build a surrogate model (an Artificial Neural Network) from a PMML file.

XIV.6.5.2 Macro Uranie

```
{

  // Create a TDataServer
  // Load a database in an ASCII file
  TDataServer * tds = new TDataServer("tdsFlowrate", "tds for flowrate");
  tds->fileDataRead("_flowrate_sampler_launcher_app_.dat");

  TCanvas *c = new TCanvas("c1", "Graph for the Macro modeler",5,64,1270,667);
  // Build a surrogate model (Artificial Neural Networks) from the PMML file
  TANNModeler* tann = new TANNModeler(tds, "uranie_ann_flowrate.pmml","ANNflowrate");

  // export the surrogate model in a C file
  tann->exportFunction("c++", "uranie_ann_flowrate_loaded","ANNflowrate");
  // load the surrogate model in the C file
  gROOT->LoadMacro("uranie_ann_flowrate_loaded.C");

  TDataServer * tdsv = new TDataServer("tdsv", "tds for surrogate model");
  tdsv->fileDataRead("_flowrate_sampler_launcher_val_.dat");

  cout << tdsv->getNPatterns() << endl;

  // evaluate the surrogate model on the database
  TLauncherFunction * tlf = new TLauncherFunction(tdsv, "ANNflowrate", "rw:r:tu:tl:hu:hl:l: ↩
      kw", "yann");
  tlf->run();

  tdsv->startViewer();
  tdsv->Draw("yann:yhat");
  //  tdsv->draw("yhat");

}
```

A TDataServer loads _flowrate_sampler_launcher_app_.dat:

```
TDataServer * tds = new TDataServer("tds", "tds for flowrate");
tds->fileDataRead("_flowrate_sampler_launcher_app_.dat");
```

The surrogate model is loaded from a PMML file:

```
TANNModeler* tann = new TANNModeler(tds, "uranie_ann_flowrate.pmml","ANNflowrate");
```

The model is exported in an external file in C++ language "uranie_ann_flowrate.C where the function name is ANNflowrate:

```
tann->exportFunction("c++", "uranie_ann_flowrate.C","ANNflowrate");
```

The model is loaded from the macro "uranie_ann_flowrate.C and applied on the database with the function ANNflowrate:

```
gROOT->LoadMacro("uranie_ann_flowrate.C");
TDataServer * tdsv = new TDataServer();
tdsv->fileDataRead("_flowrate_sampler_launcher_val_.dat");
TLauncherFunction * tlf = new TLauncherFunction(tdsv, "ANNflowrate", "rw:r:tu:tl:hu:hl:l:kw ↩
    ", "yann");
tlf->run();
```

XIV.6.5.3 Graph

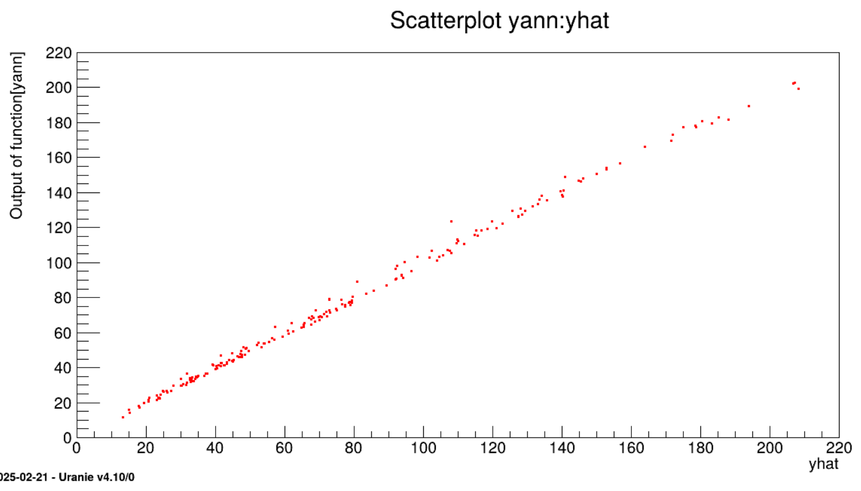


Figure XIV.68: Graph of the macro "modelerFlowrateNeuralNetworksLoadingPMML.C"

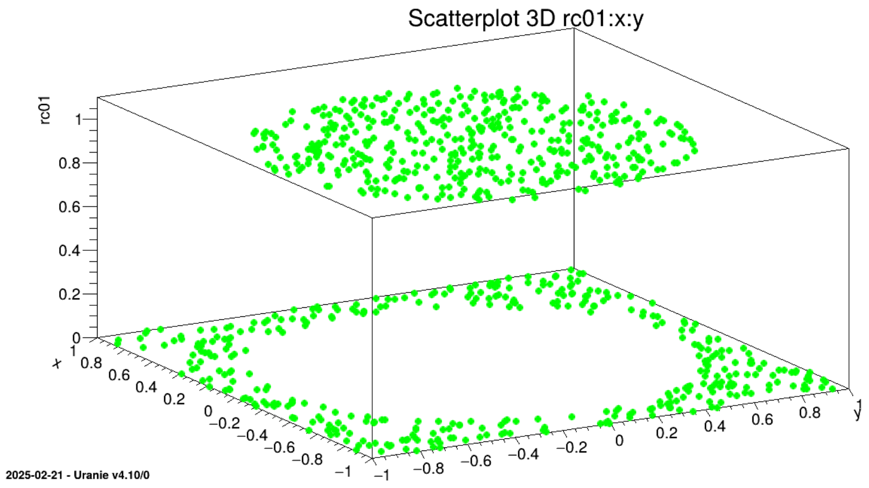
XIV.6.5.4 Console

```
Processing modelerFlowrateNeuralNetworksLoadingPMML.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

PMML Constructor: uranie_ann_flowrate.pmml

*******************************
*** TModeler::exportFunction lang[c++] file[uranie_ann_flowrate_loaded] name[ANNflowrate]  ↩
    soption[]

*******************************

*******************************
*** exportFunction lang[c++] file[uranie_ann_flowrate_loaded] name[ANNflowrate]
*** End Of exportFunction
*******************************
```

XIV.6.6 Macro "modelerClassificationNeuralNetworks.C"

XIV.6.6.1 Objective

The objective of this macro is to build a surrogate model (an Artificial Neural Network) from a database for a **"Classification"** Problem. From a first database loaded from the ASCII file `problem2Classes_001000.dat`, which defines three variables $(x,y) \in [-1.,1]^2$ and $rc01 \in \{0,1\}$ on 1000 patterns,



two ASCII files are created, one with the 800st patterns (`_problem2Classes_app_.dat`), the other with the 200 last patterns (`_problem2Classes_val_.dat`). The surrogate model is built with the database extracted from the first of the two files, the second allowing to perform calculations with a function.

XIV.6.6.2 Macro Uranie

```
{
    Int_t nH1 = 15;
    Int_t nH2 = 10;
    Int_t nH3 = 5;
    TString sX = "x:y";
    TString sY = "rc01";
    TString sYhat = Form("%shat_%d_%d_%d", sY.Data(), nH1, nH2, nH3);

    // Load a database in an ASCII file
    TDataServer * tdsu = new TDataServer("tdsu","tds u");
    tdsu->fileDataRead("problem2Classes_001000.dat");

    // Split into 2 datasets for learning and testing
    tdsu->exportData("_problem2Classes_app_.dat", "", Form("%s<=800", tdsu->getIteratorName ↩
        ()));
    tdsu->exportData("_problem2Classes_val_.dat", "", Form("%s>800", tdsu->getIteratorName ↩
        ()));

    TDataServer * tds = new TDataServer("tdsApp", "tds App for problem2Classes");
    tds->fileDataRead("_problem2Classes_app_.dat");

    TANNModeler* tann = new TANNModeler(tds, Form("%s,%d,%d,%d,@%s", sX.Data(), nH1, nH2,  ↩
        nH3, sY.Data()));
    //tann->setLog();
    tann->setNormalization(TANNModeler::kMinusOneOne);
    tann->setFcnTol(1e-6);
    tann->train(2, 2, "test", kFALSE);

    tann->exportFunction("c++","uranie_ann_problem2Classes","ANNproblem2Classes");

    gROOT->LoadMacro("uranie_ann_problem2Classes.C");

    TDataServer * tdsv = new TDataServer();
    tdsv->fileDataRead("_problem2Classes_val_.dat");

    // evaluate the surrogate model on the database
    TLauncherFunction * tlf = new TLauncherFunction(tdsv, "ANNproblem2Classes", sX, sYhat);
    tlf->run();

    TCanvas *c = new TCanvas("c1", "Graph for the Macro modeler",5,64,1270,667);
    // Buils a surrogate model (Artificial Neural Networks) from the DataBase
    tds->getTuple()->SetMarkerStyle(8);
    tds->getTuple()->SetMarkerSize(1.0);
    tds->getTuple()->SetMarkerColor(kGreen);
    tds->draw(Form("%s:%s", sY.Data(), sX.Data()));

    tdsv->getTuple()->SetMarkerStyle(8);
    tdsv->getTuple()->SetMarkerSize(0.75);
    tdsv->getTuple()->SetMarkerColor(kRed);
    tdsv->Draw(Form("%s:%s", sYhat.Data(), sX.Data()),"","same");

}
```

The main `TDataServer` loads the main ASCII data file `problem2Classes_001000.dat`

```
TDataServer * tdsu = new TDataServer("tdsu","tds u");
tdsu->fileDataRead("problem2Classes_001000.dat");
```

The database is split with the internal iterator attribute in two parts by exporting the 800st patterns in a file and the remaining 200 in another one

```
tdsu->exportData("_problem2Classes_app_.dat", "",Form("%s<=800", tdsu->getIteratorName()));
tdsu->exportData("_problem2Classes_val_.dat", "", Form("%s>800", tdsu->getIteratorName()));
```

A second TDataServer loads _problem2Classes_app_.dat and builds the surrogate model over all the variables with 3 hidden layers, a **Hyperbolic Tangent (TanH)** activation function (normalization) in the 3 hidden layers and set the function tolerance to 1e-.6. The **"@"** character behind the output name defines a **classification** problem.

```
TDataServer * tds = new TDataServer("tds", "tds for the 2 Classes problem");
tds->fileDataRead("_problem2Classes_app_.dat");
TANNModeler* tann = new TANNModeler(tds, Form("%s,%d,%d,%d,@%s", sX.Data(), nH1, nH2, nH3,  ↩
    sY.Data()));
tann->setNormalization(TANNModeler::kMinusOneOne);
tann->setFcnTol(1e-6);
tann->train(3, 2, "test");
```

The model is exported in an external file in C++ language "uranie_ann_problem2Classes.C" where the function name is ANNproblem2Classes

```
tann->exportFunction("c++", "uranie_ann_problem2Classes.C","ANNproblem2Classes");
```

The model is loaded from the macro "uranie_ann_problem2Classes.C and applied on the second database with the function ANNproblem2Classes.

```
gROOT->LoadMacro("uranie_ann_problem2Classes.C");
TDataServer * tdsv = new TDataServer();
tdsv->fileDataRead("_problem2Classes_val_.dat");
TLauncherFunction * tlf = new TLauncherFunction(tdsv, "ANNproblem2Classes", sX, sYhat);
tlf->run();
```

We draw on a 3D graph, the learning database (in green) and the estimations by the Artificial Neural Network with red points.

```
TCanvas *c = new TCanvas("c1", "Graph for the Macro modeler",5,64,1270,667);
tds->getTuple()->SetMarkerStyle(8);
tds->getTuple()->SetMarkerSize(1.0);
tds->getTuple()->SetMarkerColor(kGreen);
tds->draw(Form("%s:%s", sY.Data(), sX.Data()));

tdsv->getTuple()->SetMarkerStyle(8);
tdsv->getTuple()->SetMarkerSize(0.75);
tdsv->getTuple()->SetMarkerColor(kRed);
tdsv->Draw(Form("%s:%s", sYhat.Data(), sX.Data()),"","same");
```

XIV.6.6.3 Graph

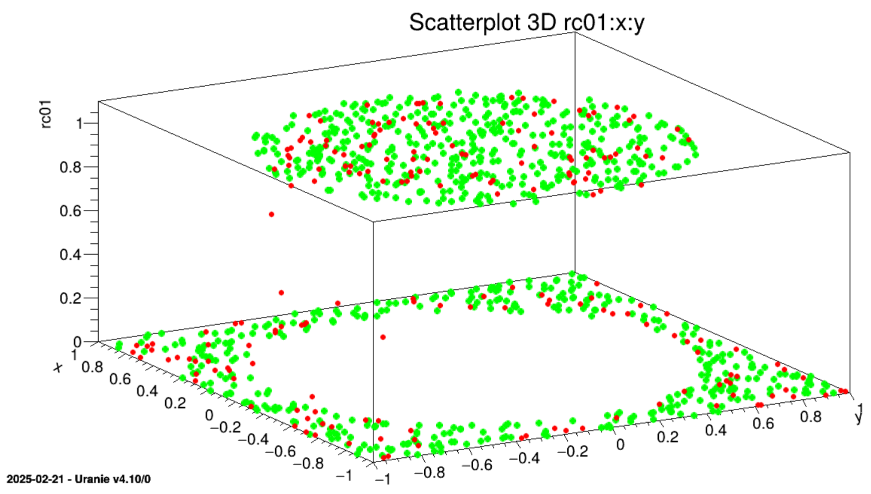


Figure XIV.69: Graph of the macro "`modelerClassificationNeuralNetworks.C`"

XIV.6.6.4 Console

```
Processing modelerClassificationNeuralNetworks.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[759]
 <URANIE::WARNING> TDataServer::fileDataRead: Expected iterator tdsu__n__iter__ not found ↩
    but _tds___n__iter__ looks like an URANIE iterator => Will be used as so.
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[759]
 <URANIE::WARNING> TDataServer::fileDataRead: Expected iterator tdsApp__n__iter__ not found ↩
     but _tds___n__iter__ looks like an URANIE iterator => Will be used as so.
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 ** TANNModeler::train niter[2] ninit[2]
 ** init the ANN
 ** Input  (1) Name[x] Min[-0.999842] Max[0.998315]
 ** Input  (2) Name[y] Min[-0.999434] Max[0.998071]
 ** Output  (1) Name[rc01] Min[0] Max[1]
 ** Tolerance  (1e-06)
 ** sHidden    (15,10,5) (30)
 ** Nb Weights (266)
 ** ActivationFunction[TanH]
 **

 ** iter[1/2] : ** : mse_min[0.00813999]
 ** iter[2/2] : ** : mse_min[0.00636923]
 ** solutions : 2
 ** isol[1] iter[0] learn[0.00405863] test[0.00813999] *
 ** isol[2] iter[1] learn[0.00560685] test[0.00636923] *
 ** CPU training finished. Total elapsed time: 17 sec

*******************************
*** TModeler::exportFunction lang[c++] file[uranie_ann_problem2Classes] name[ ↩
    ANNproblem2Classes] soption[]

*******************************

*******************************
*** exportFunction lang[c++] file[uranie_ann_problem2Classes] name[ANNproblem2Classes]
*** End Of exportFunction
*******************************
```

XIV.6.7 Macro "modelerFlowratePolynChaosRegression.C"

XIV.6.7.1 Objective

The objective of this macro is to build a polynomial chaos expansion in order to get a surrogate model along with a global sensitivity interpretation for the `flowrate` function, whose purpose and behaviour have been already introduced in Section IV.1.2.1. The method used here is the regression one, as discussed below.

XIV.6.7.2 Macro Uranie

```
{

    gROOT->LoadMacro("UserFunctions.C");

    // Define the DataServer
    TDataServer *tds = new TDataServer("tdsFlowrate", "Design of experiments for Flowrate") ↩
        ;
    // Add the eight attributes of the study with uniform law
    tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
    tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
    tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
    tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
    tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
    tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
    tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
    tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

    // Define of TNisp object
    TNisp *nisp = new TNisp(tds);
    nisp->generateSample("QmcSobol", 500); //State that there is a sample ...

    // Compute the output variable
    TLauncherFunction * tlf = new TLauncherFunction(tds, "flowrateModel","*","ymod");
    tlf->setDrawProgressBar(kFALSE);
    tlf->run();

    // build a chaos polynomial
    TPolynomialChaos *pc = new TPolynomialChaos(tds,nisp);

    // compute the pc coefficients using the "Regression" method
    Int_t degree = 4;
    pc->setDegree(degree);
    pc->computeChaosExpansion("Regression");

    // Uncertainty and sensitivity analysis
    cout << "Variable ymod ================" << endl;
    cout << "Mean     = " << pc->getMean("ymod") << endl;
    cout << "Variance = " << pc->getVariance("ymod") << endl;
    cout << "First Order Indices ================" << endl;
    cout << "Indice First Order[1] = " <<  pc->getIndexFirstOrder(0,0) << endl;
    cout << "Indice First Order[2] = " <<  pc->getIndexFirstOrder(1,0) << endl;
    cout << "Indice First Order[3] = " <<  pc->getIndexFirstOrder(2,0) << endl;
    cout << "Indice First Order[4] = " <<  pc->getIndexFirstOrder(3,0) << endl;
    cout << "Indice First Order[5] = " <<  pc->getIndexFirstOrder(4,0) << endl;
    cout << "Indice First Order[6] = " <<  pc->getIndexFirstOrder(5,0) << endl;
    cout << "Indice First Order[7] = " <<  pc->getIndexFirstOrder(6,0) << endl;
    cout << "Indice First Order[8] = " <<  pc->getIndexFirstOrder(7,0) << endl;
    cout << "Total Order Indices ================" << endl;
    cout << "Indice Total Order[1] = " << pc->getIndexTotalOrder("rw","ymod") << endl;
    cout << "Indice Total Order[2] = " << pc->getIndexTotalOrder("r","ymod") << endl;
    cout << "Indice Total Order[3] = " << pc->getIndexTotalOrder("tu","ymod") << endl;
    cout << "Indice Total Order[4] = " << pc->getIndexTotalOrder("tl","ymod") << endl;
    cout << "Indice Total Order[5] = " << pc->getIndexTotalOrder("hu","ymod") << endl;
    cout << "Indice Total Order[6] = " << pc->getIndexTotalOrder("hl","ymod") << endl;
    cout << "Indice Total Order[7] = " << pc->getIndexTotalOrder("l","ymod") << endl;
    cout << "Indice Total Order[8] = " << pc->getIndexTotalOrder("kw","ymod") << endl;

    // Dump main factors up to a certain threshold
    Double_t seuil = 0.99;
    cout<<"Ordered functionnal ANOVA"<<endl;
    pc->getAnovaOrdered(seuil,0);

    cout << "Number of experiments = " << tds->getNPatterns() << endl;

    //save the pv in a program (C langage)
    pc->exportFunction("NispFlowrate","NispFlowrate");

}
```

The first part is just creating a `TDataServer` and providing the attributes needed to define the problem. From there, a `TNisp` object is created, providing the dataserver that specifies the inputs. This class is used to generate the sample.

```
// Define of TNisp object
TNisp *nisp = new TNisp(tds);
nisp->generateSample("QmcSobol", 500); //State that there is a sample ...
```

The function is launched through a `TLauncherFunction` instance in order to get the output values that will be needed to train the surrogate model.

```
TLauncherFunction * tlf = new TLauncherFunction(tds, "flowrateModel","*","ymod");
tlf->run();
```

Finally, a `TPolynomialChaos` instance is created and the computation of the coefficients is performed by requesting a truncature on the resulting degree of the polynomial expansion (set to 4) and the use of a regression method.

```
// build a chaos polynomial
TPolynomialChaos *pc = new TPolynomialChaos(tds,nisp);
// compute the pc coefficients using the "Regression" method
Int_t degree = 4;
pc->setDegree(degree);
pc->computeChaosExpansion("Regression");
```

The rest of the code is showing how to access the resulting sensitivity indices either one-by-one, or ordered up to a chosen threshold of the output variance.

XIV.6.7.3 Console

```
Processing modelerFlowratePolynChaosRegression.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

Variable ymod ================
Mean     = 77.651
Variance = 2079.68
First Order Indices ================
Indice First Order[1] = 0.828732
Indice First Order[2] = 1.05548e-06
Indice First Order[3] = 9.86805e-08
Indice First Order[4] = 4.77668e-06
Indice First Order[5] = 0.0414015
Indice First Order[6] = 0.0413215
Indice First Order[7] = 0.039401
Indice First Order[8] = 0.00954644
Total Order Indices ================
Indice Total Order[1] = 0.866773
Indice Total Order[2] = 9.27467e-06
Indice Total Order[3] = 7.01768e-06
Indice Total Order[4] = 1.71944e-05
Indice Total Order[5] = 0.0541916
Indice Total Order[6] = 0.0540578
Indice Total Order[7] = 0.0522121
Indice Total Order[8] = 0.0127809
```

XIV.6.8 Macro "modelerFlowratePolynChaosIntegration.C"

XIV.6.8.1 Objective

The objective of this macro is to build a polynomial chaos expansion in order to get a surrogate model along with a global sensitivity interpretation for the `flowrate` function, whose purpose and behaviour have been already introduced in Section IV.1.2.1. The method used here is the regression one, as discussed below.

XIV.6.8.2 Macro Uranie

```
{

    gROOT->LoadMacro("UserFunctions.C");

    // Define the DataServer
    TDataServer *tds = new TDataServer("tdsFlowrate", "Design of experiments for Flowrate") ↩
        ;
    // Add the eight attributes of the study with uniform law
    tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
    tds->addAttribute( new TUniformDistribution("r", 100.0, 50000.0));
    tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
    tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
    tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
    tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
    tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
    tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

    // Define of TNisp object
    TNisp *nisp = new TNisp(tds);
    nisp->generateSample("Petras", 5); //State that there is a sample ...

    // Compute the output variable
    TLauncherFunction * tlf = new TLauncherFunction(tds, "flowrateModel","*","ymod");
    tlf->setDrawProgressBar(kFALSE);
    tlf->run();

    // build a chaos polynomial
    TPolynomialChaos *pc = new TPolynomialChaos(tds,nisp);

    // compute the pc coefficients using the "Integration" method
    Int_t degree = 4;
    pc->setDegree(degree);
    pc->computeChaosExpansion("Integration");

    // Uncertainty and sensitivity analysis
    cout << "Variable ymod ================" << endl;
    cout << "Mean     = " << pc->getMean("ymod") << endl;
    cout << "Variance = " << pc->getVariance("ymod") << endl;
    cout << "First Order Indices ================" << endl;
    cout << "Indice First Order[1] = " <<  pc->getIndexFirstOrder(0,0) << endl;
    cout << "Indice First Order[2] = " <<  pc->getIndexFirstOrder(1,0) << endl;
    cout << "Indice First Order[3] = " <<  pc->getIndexFirstOrder(2,0) << endl;
    cout << "Indice First Order[4] = " <<  pc->getIndexFirstOrder(3,0) << endl;
    cout << "Indice First Order[5] = " <<  pc->getIndexFirstOrder(4,0) << endl;
    cout << "Indice First Order[6] = " <<  pc->getIndexFirstOrder(5,0) << endl;
    cout << "Indice First Order[7] = " <<  pc->getIndexFirstOrder(6,0) << endl;
    cout << "Indice First Order[8] = " <<  pc->getIndexFirstOrder(7,0) << endl;
    cout << "Total Order Indices ================" << endl;
    cout << "Indice Total Order[1] = " << pc->getIndexTotalOrder("rw","ymod") << endl;
    cout << "Indice Total Order[2] = " << pc->getIndexTotalOrder("r","ymod") << endl;
    cout << "Indice Total Order[3] = " << pc->getIndexTotalOrder("tu","ymod") << endl;
    cout << "Indice Total Order[4] = " << pc->getIndexTotalOrder("tl","ymod") << endl;
    cout << "Indice Total Order[5] = " << pc->getIndexTotalOrder("hu","ymod") << endl;
    cout << "Indice Total Order[6] = " << pc->getIndexTotalOrder("hl","ymod") << endl;
    cout << "Indice Total Order[7] = " << pc->getIndexTotalOrder("l","ymod") << endl;
    cout << "Indice Total Order[8] = " << pc->getIndexTotalOrder("kw","ymod") << endl;

    // Dump main factors up to a certain threshold
    Double_t seuil = 0.99;
    cout<<"Ordered functionnal ANOVA"<<endl;
    pc->getAnovaOrdered(seuil,0);

    cout << "Number of experiments = " << tds->getNPatterns() << endl;

    //save the pv in a program (C langage)
    pc->exportFunction("NispFlowrate","NispFlowrate");

}
```

The first part is just creating a `TDataServer` and providing the attributes needed to define the problem. From there, a `TNisp` object is created, providing the dataserver that specifies the inputs. This class is used to generate the sample (Petras being a design-of-experiments dedicated to integration problem).

```
// Define of TNisp object
TNisp *nisp = new TNisp(tds);
nisp->generateSample("Petras", 5); //State that there is a sample ...
```

The function is launched through a `TLauncherFunction` instance in order to get the output values that will be needed to train the surrogate model.

```
TLauncherFunction * tlf = new TLauncherFunction(tds, "flowrateModel","*","ymod");
tlf->run();
```

Finally, a `TPolynomialChaos` instance is created and the computation of the coefficients is performed by requesting a truncature on the resulting degree of the polynomial expansion (set to 4) and the use of a regression method.

```
// build a chaos polynomial
TPolynomialChaos *pc = new TPolynomialChaos(tds,nisp);
// compute the pc coefficients using the "Integration" method
Int_t degree = 4;
pc->setDegree(degree);
pc->computeChaosExpansion("Integration");
```

The rest of the code is showing how to access the resulting sensitivity indices either one-by-one, or ordered up to a chosen threshold of the output variance.

XIV.6.8.3 Console

```
Processing modelerFlowratePolynChaosIntegration.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

Variable ymod ================
Mean     = 77.6511
Variance = 2078.91
First Order Indices ================
Indice First Order[1] = 0.828922
Indice First Order[2] = 1.04477e-06
Indice First Order[3] = 8.94692e-12
Indice First Order[4] = 5.30299e-06
Indice First Order[5] = 0.0413852
Indice First Order[6] = 0.0413852
Indice First Order[7] = 0.0393419
Indice First Order[8] = 0.00952157
Total Order Indices ================
Indice Total Order[1] = 0.866834
Indice Total Order[2] = 2.16178e-06
Indice Total Order[3] = 2.28296e-09
Indice Total Order[4] = 1.07386e-05
Indice Total Order[5] = 0.0541095
Indice Total Order[6] = 0.0541095
Indice Total Order[7] = 0.0520767
Indice Total Order[8] = 0.0127306
```

XIV.6.9 Macro "modelerbuildSimpleGP.C"

XIV.6.9.1 Objective

This macro is the one described in Section V.6.2.2, that creates a simple gaussian process, whose training (utf_4D_train.dat) and testing (utf_4D_test.dat) database can both be found in the document folder of the Uranie installation (${URANIESYS}/share/uranie/docUMENTS).

XIV.6.9.2 Macro Uranie

```
{
  // Load observations
  TDataServer *tdsObs = new TDataServer("tdsObs", "observations");
  tdsObs->fileDataRead("utf_4D_train.dat");

  // Construct the GPBuilder
  TGPBuilder *gpb = new TGPBuilder(tdsObs,            // observations data
           "x1:x2:x3:x4",       // list of input variables
           "y",                 // output variable
           "matern3/2");        // name of the correlation function


  // Search for the optimal hyper-parameters
  gpb->findOptimalParameters("ML",            // optimisation criterion
           100,              // screening design size
           "neldermead",     // optimisation algorithm
           500);             // max. number of optimisation iterations

  // Construct the kriging model
  TKriging *krig = gpb->buildGP();

  // Display model information
  krig->printLog();

  }
```

XIV.6.9.3 Console

This is the result of the last command:

```
*******************************
** TKriging::printLog[]
*******************************
 Input Variables:      x1:x2:x3:x4
 Output Variable:      y
 Deterministic trend:
 Correlation function: URANIE::Modeler::TMatern32CorrFunction
 Correlation length:   normalised   (not normalised)
                       1.6181e+00 (1.6172e+00 )
                       1.4372e+00 (1.4370e+00 )
                       1.5026e+00 (1.5009e+00 )
                       6.7884e+00 (6.7944e+00 )

 Variance of the gaussian process:      70.8755
 RMSE (by Leave One Out):               0.499108
 Q2:                                    0.849843
*******************************
```

XIV.6.10 Macro "modelerbuildGPInitPoint.C"

XIV.6.10.1 Objective

This macro is the one described in Section V.6.2.7, that creates a gaussian process, whose training (utf_4D_train.dat) and testing (utf_4D_test.dat) database can both be found in the document folder of the Uranie installation (${URANIESYS}/share/uranie/docUMENTS), and whose starting point, in the parameter space, is specified.

XIV.6.10.2 Macro Uranie

```
{
 // Load observations
 TDataServer *tdsObs = new TDataServer("tdsObs", "observations");
 tdsObs->fileDataRead("utf_4D_train.dat");

 // Construct the GPBuilder
 TGPBuilder *gpb = new TGPBuilder(tdsObs, "x1:x2:x3:x4", "y", "matern3/2");

 // Set the correlation function parameters
 Double_t params[4] = {1.0, 0.25, 0.01, 0.3};
 gpb->getCorrFunction()->setParameters(params);

 // Find the optimal parameters
 gpb->findOptimalParameters("ML",          // optimisation criterion
                            0,             // screening size MUST be equal to 0
          "neldermead", // optimisation algorithm
          500,          // max. number of optimisation iterations
          kFALSE);      // we don't reset the parameters of the GP builder

  // Create the kriging model
  TKriging *krig = gpb->buildGP();

  // Display model information
  krig->printLog();

  }
```

XIV.6.10.3 Console

This is the result of the last command:

```
*******************************
** TKriging::printLog[]
*******************************
 Input Variables:      x1:x2:x3:x4
 Output Variable:      y
 Deterministic trend:
 Correlation function: URANIE::Modeler::TMatern32CorrFunction
 Correlation length:   normalised   (not normalised)
                       1.6182e+00 (1.6173e+00 )
                       1.4373e+00 (1.4371e+00 )
                       1.5027e+00 (1.5011e+00 )
                       6.7895e+00 (6.7955e+00 )

 Variance of the gaussian process:      70.8914
 RMSE (by Leave One Out):               0.49911
 Q2:                                      0.849842
*******************************
```

XIV.6.11 Macro "modelerbuildGPWithAPriori.C"

XIV.6.11.1 Objective

This macro is the one described in Section V.6.2.4, that creates a gaussian process with a specific trend and an *a priori* knowledge on the mean and variance of the trend parameters.

XIV.6.11.2 Macro Uranie

```
{
    // Load observations
   TDataServer *tdsObs = new TDataServer("tdsObs", "observations");
   tdsObs->fileDataRead("utf_4D_train.dat");

   // Construct the GPBuilder
   TGPBuilder *gpb = new TGPBuilder(tdsObs,             // observations data
            "x1:x2:x3:x4",      // list of input variables
            "y",                // output variable
            "matern3/2",        // name of the correlation function
            "linear");          // trend defined by a keyword


   // Bayesian study
   Double_t meanPrior[5] = {0.0, 0.0, -1.0, 0.0, -0.1};
   Double_t covPrior[25] = {1e-4, 0.0 , 0.0 , 0.0 , 0.0 ,
          0.0 , 1e-4, 0.0 , 0.0 , 0.0 ,
          0.0 , 0.0 , 1e-4, 0.0 , 0.0 ,
          0.0 , 0.0 , 0.0 , 1e-4, 0.0 ,
          0.0 , 0.0 , 0.0 , 0.0 , 1e-4};
   gpb->setPriorData(meanPrior, covPrior);

   // Search for the optimal hyper-parameters
   gpb->findOptimalParameters("ReML",          // optimisation criterion
            100,              // screening design size
            "neldermead",     // optimisation algorithm
            500);             // max. number of optimisation iterations

   // Construct the kriging model
   TKriging *krig = gpb->buildGP();

   // Display model information
   krig->printLog();

  }
```

XIV.6.11.3 Console

This is the result of the last command:

```
*******************************
** TKriging::printLog[]
*******************************
```

```
 Input Variables:      x1:x2:x3:x4
 Output Variable:      y
 Deterministic trend:  linear
 Trend parameters (5): [3.06586494e-05; 1.64887174e-05; -9.99986787e-01; 1.51959859e-05;  ↩
     -9.99877606e-02 ]
 Correlation function: URANIE::Modeler::TMatern32CorrFunction
 Correlation length:   normalised   (not normalised)
                       2.1450e+00 (2.1438e+00 )
                       1.9092e+00 (1.9090e+00 )
                       2.0062e+00 (2.0040e+00 )
                       8.4315e+00 (8.4390e+00 )

 Variance of the gaussian process:      155.533
 RMSE (by Leave One Out):               0.495448
 Q2:                                    0.852037
*******************************
```

XIV.6.12 Macro "modelerbuildSimpleGPEstim.C"

XIV.6.12.1 Objective

This macro is the one described in Section V.6.3.1, to create and use a simple gaussian process, whose training (`utf_4D_train.dat`) and testing (`utf_4D_test.dat`) database can both be found in the document folder of the Uranie installation (`${URANIESYS}/share/uranie/docUMENTS`). It uses the simple one-by-one approch described in the [30] for completness.

XIV.6.12.2 Macro Uranie

```
{
  // Load observations
  TDataServer *tdsObs = new TDataServer("tdsObs", "observations");
  tdsObs->fileDataRead("utf_4D_train.dat");

  // Construct the GPBuilder
  TGPBuilder *gpb = new TGPBuilder(tdsObs,            // observations data
           "x1:x2:x3:x4",     // list of input variables
           "y",               // output variable
           "matern3/2");       // name of the correlation function


  // Search for the optimal hyper-parameters
  gpb->findOptimalParameters("ML",            // optimisation criterion
           100,             // screening design size
           "neldermead",    // optimisation algorithm
           500);            // max. number of optimisation iterations

  // Construct the kriging model
  TKriging *krig = gpb->buildGP();

  // Display model information
  krig->printLog();

  // Load the data to estimate
  TDataServer *tdsEstim = new TDataServer("tdsEstim", "estimations");
  tdsEstim->fileDataRead("utf_4D_test.dat");

  // Construction of the launcher
  TLauncher2 *lanceur = new TLauncher2(tdsEstim,          // data to estimate
                krig,               // model used for the estimation
                "x1:x2:x3:x4",      // list of the input variables
                "yEstim:vEstim");   // name given to the model's outputs

  // Launch the estimations
  lanceur->solverLoop();

  // Display some results
  tdsEstim->draw("yEstim:y");

  }
```

XIV.6.12.3 Graph

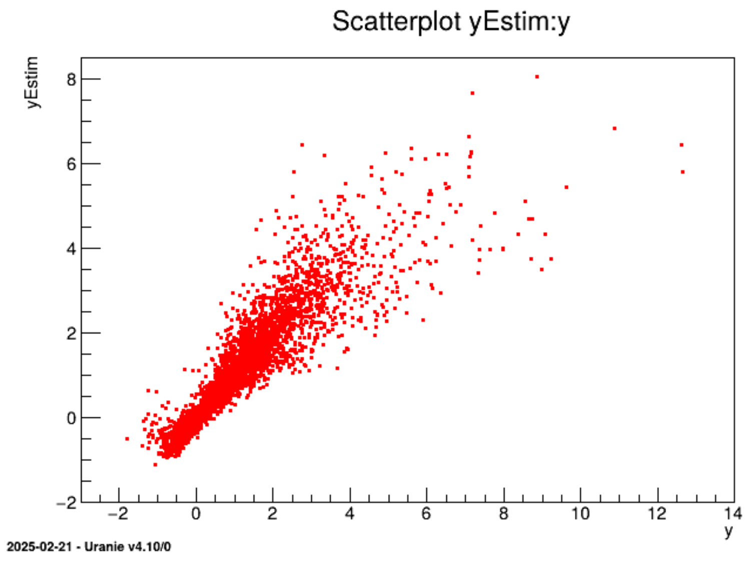


Figure XIV.70: Graph of the macro "modelerbuildSimpleGPEstim.C"

XIV.6.13 Macro "modelerbuildSimpleGPEstimWithCov.C"

XIV.6.13.1 Objective

This macro is the one described in Section V.6.3.2, to create and use a simple gaussian process, whose training (utf_4D_train.dat) and testing (utf_4D_test.dat) database can both be found in the document folder of the Uranie installation (${URANIESYS}/share/uranie/docUMENTS). It uses the global approach, computing the input covariance matrix and translating it to the prediction covariance matrix.

XIV.6.13.2 Macro Uranie

```
{
  // Load observations
  TDataServer *tdsObs = new TDataServer("tdsObs", "observations");
  tdsObs->fileDataRead("utf_4D_train.dat");
  // Construct the GPBuilder
  TGPBuilder *gpb = new TGPBuilder(tdsObs,            // observations data
           "x1:x2:x3:x4",     // list of input variables
           "y",               // output variable
           "matern1/2");       // name of the correlation function


  // Search for the optimal hyper-parameters
  gpb->findOptimalParameters("ML",            // optimisation criterion
           100,             // screening design size
           "neldermead",    // optimisation algorithm
           500);            // max. number of optimisation iterations

  // Construct the kriging model
  TKriging *krig = gpb->buildGP();

  // Display model information
  krig->printLog();

  // Load the data to estimate
  TDataServer *tdsEstim = new TDataServer("tdsEstim", "estimations");
  tdsEstim->fileDataRead("utf_4D_test.dat");

  // Reducing the database to 1000 first event (prediction cov matrix of a million value !)
  int nST=1000;
  tdsEstim->exportData("utf_4D_test_red.dat","",Form("tdsEstim__n__iter__<=%d",nST));
  tdsEstim->fileDataRead("utf_4D_test_red.dat",false,true); // Reload reduce sample

 krig->estimateWithCov(tdsEstim, // data to estimate
                       "x1:x2:x3:x4",// list of the input variables
                       "yEstim:vEstim", // name given to the model's outputs
                       "y", //name of the true reference if validation database
                       ""); //options

  TCanvas *c2=NULL;
  // Residuals plots if true information provided
  if( tdsEstim->isAttribute("_Residuals_") )
  {
      c2 = new TCanvas("c2","c2",1200,800);
      c2->Divide(2,1);
      c2->cd(1);
      // Usual residual considering uncorrated input points
      tdsEstim->Draw("_Residuals_");
      c2->cd(2);
      // Corrected residuals, with prediction covariance matrix
      tdsEstim->Draw("_uncorrResiduals_");
  }

  // Retrieve all the prediction covariance coefficient
  tdsEstim->getTuple()->SetEstimate( nST * nST ); //allocate the correct size
  // Get a pointer to all values
  tdsEstim->getTuple()->Draw("_CovarianceMatrix_","","goff");
  double *cov=tdsEstim->getTuple()->GetV1();

  //Put these in a matrix nicely created
  TMatrixD Cov(nST,nST);
  Cov.Use(0,nST-1,0,nST-1,cov);

  //Print it if size is reasonnable
  if(nST<10)
      Cov.Print();
```

```
}
```

XIV.6.13.3 Graph

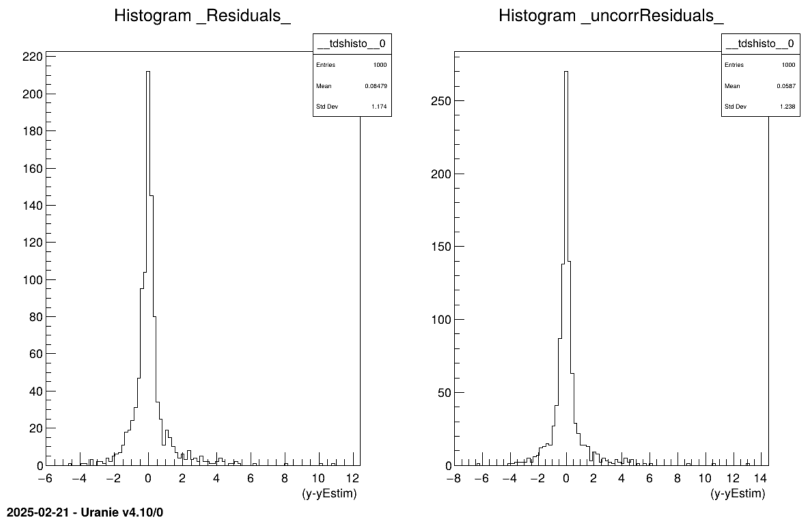


Figure XIV.71: Graph of the macro "modelerbuildSimpleGPEstimWithCov.C"

XIV.6.14 Macro "modelerTestKriging.C"

XIV.6.14.1 Objective

The idea is to provide a simple example of a kriging usage, and an how to, to produce plots in one dimension to represent the results. This example is the one taken from [30] that uses a very simple set of six points as training database:

```
#TITLE: utf-1D-train.dat
#COLUMN_NAMES: x1|y

6.731290e-01 3.431918e-01
7.427596e-01 9.356860e-01
4.518467e-01 -3.499771e-01
2.215734e-02 2.400531e+00
9.915253e-01 2.412209e+00
1.815769e-01 1.589435e+00
```

The aim will be to get a kriging model that describes this dataset and to check its consistency over a certain number of points (here 100 points) which will be the testing database:

```
#TITLE: utf-1D-test.dat
#COLUMN_NAMES: x1|y

5.469435e-02 2.331524e+00
3.803054e-01 -3.277316e-02
7.047152e-01 6.030177e-01
2.360045e-02 2.398694e+00
9.271965e-01 2.268814e+00
7.868263e-01 1.324318e+00
```

```
7.791920e-01 1.257942e+00
6.107965e-01 -8.514510e-02
1.362316e-01 1.926999e+00
5.709913e-01 -2.758435e-01
8.738804e-01 1.992941e+00
2.251602e-01 1.219826e+00
9.175259e-01 2.228545e+00
5.128368e-01 -4.096161e-01
7.692913e-01 1.170999e+00
7.394406e-01 9.062407e-01
5.364506e-01 -3.772856e-01
1.861864e-01 1.551961e+00
7.573444e-01 1.065237e+00
1.005755e-01 2.141109e+00
9.114685e-01 2.201001e+00
3.628465e-01 7.920271e-02
2.383583e-01 1.103353e+00
7.468092e-01 9.716492e-01
3.126209e-01 4.578112e-01
8.034716e-01 1.466248e+00
6.730402e-01 3.424931e-01
8.021345e-01 1.455015e+00
2.503736e-01 9.966807e-01
9.001793e-01 2.145059e+00
7.019990e-01 5.799112e-01
6.001746e-01 -1.432102e-01
4.925013e-01 -4.126441e-01
5.685795e-01 -2.849419e-01
1.238257e-01 2.007351e+00
2.825838e-01 7.124861e-01
4.025708e-01 -1.574002e-01
8.562999e-01 1.875879e+00
3.214125e-01 3.865241e-01
2.021767e-01 1.418581e+00
6.338581e-01 5.717657e-02
3.042007e-01 5.276410e-01
4.860707e-01 -4.088007e-01
9.645326e-01 2.379243e+00
3.583711e-02 2.378513e+00
2.143110e-01 1.314473e+00
7.299624e-01 8.224203e-01
2.719263e-02 2.393622e+00
3.321495e-01 3.020224e-01
8.642671e-01 1.930341e+00
8.893604e-01 2.086039e+00
1.119469e-01 2.078562e+00
9.859741e-01 2.408725e+00
5.594688e-01 -3.166326e-01
1.904448e-01 1.516930e+00
4.232618e-01 -2.529865e-01
1.402221e-01 1.899932e+00
2.647519e-01 8.691058e-01
1.667035e-01 1.706823e+00
2.332246e-01 1.148786e+00
8.324190e-01 1.700059e+00
4.743443e-01 -3.958790e-01
3.435927e-01 2.154677e-01
9.846049e-01 2.407603e+00
9.705327e-01 2.390043e+00
6.631883e-01 2.662970e-01
6.153726e-01 -5.862472e-02
4.632361e-01 -3.766509e-01
```

```
6.474053e-01 1.502050e-01
7.161034e-02 2.273461e+00
4.514511e-01 -3.489255e-01
5.976782e-02 2.315661e+00
8.361934e-01 1.729000e+00
5.280981e-01 -3.922313e-01
9.394759e-01 2.313181e+00
2.710088e-01 8.138628e-01
8.161943e-01 1.571375e+00
5.047683e-01 -4.135789e-01
8.427635e-02 2.220534e+00
3.540224e-01 1.400987e-01
4.698548e-03 2.413597e+00
9.124315e-02 2.188105e+00
9.996285e-01 2.414210e+00
4.167139e-01 -2.249546e-01
5.892062e-01 -1.978247e-01
2.929336e-01 6.231119e-01
4.456454e-01 -3.325379e-01
1.148699e-02 2.410532e+00
3.892636e-01 -8.548741e-02
7.188374e-01 7.248622e-01
3.697949e-01 3.323350e-02
6.864519e-01 4.502113e-01
1.586679e-01 1.767741e+00
6.603030e-01 2.445009e-01
6.277168e-01 1.721489e-02
4.305704e-01 -2.817686e-01
1.553435e-01 1.792379e+00
5.476842e-01 -3.512131e-01
8.475444e-01 1.813503e+00
9.527370e-01 2.352313e+00
```

In this example, two different correlation functions are tested and the obtained results are compared at the end.

XIV.6.14.2 Macro Uranie

```
void PrintSolutions(TDataServer *tdsObs, TDataServer *tdsEstim, string title)
{

  const int nbObs = tdsObs->getNPatterns();
  const int nbEst = tdsEstim->getNPatterns();
  vector<double> Rxarr(nbEst,0), x_arr(nbEst,0), y_est(nbEst,0), y_rea(nbEst,0), std_var( ↩
      nbEst,0);

  tdsEstim->computeRank("x1");
  tdsEstim->getTuple()->copyBranchData(&Rxarr[0],nbEst,"Rk_x1");
  tdsEstim->getTuple()->copyBranchData(&x_arr[0],nbEst,"x1");
  tdsEstim->getTuple()->copyBranchData(&y_est[0],nbEst,"yEstim");
  tdsEstim->getTuple()->copyBranchData(&y_rea[0],nbEst,"y");
  tdsEstim->getTuple()->copyBranchData(&std_var[0],nbEst,"vEstim");

  vector<double> xarr(nbEst,0), yest(nbEst,0), yrea(nbEst,0), stdvar(nbEst,0);
  int ind;
  for (int i=0; i<nbEst; i++)
  {
    ind = int(Rxarr[i]) - 1;
    xarr[ind] = x_arr[i];
    yest[ind] = y_est[i];
    yrea[ind] = y_rea[i];
    stdvar[ind] = 2*sqrt(fabs(std_var[i]));
  }

  vector<double> xobs(nbObs,0); tdsObs->getTuple()->copyBranchData(&xobs[0],nbObs,"x1");
  vector<double> yobs(nbObs,0); tdsObs->getTuple()->copyBranchData(&yobs[0],nbObs,"y");

  TGraph *gobs = new TGraph(nbObs,&xobs[0],&yobs[0]);
  gobs->SetMarkerColor(1); gobs->SetMarkerSize(1); gobs->SetMarkerStyle(8);
  TGraphErrors *gest = new TGraphErrors(nbEst,&xarr[0],&yest[0],0,&stdvar[0]);
  gest->SetLineColor(2); gest->SetLineWidth(1); gest->SetFillColor(2); gest->SetFillStyle ↩
      (3002);
  TGraph *grea = new TGraph(nbEst,&xarr[0],&yrea[0]);
  grea->SetMarkerColor(4); grea->SetMarkerSize(1); grea->SetMarkerStyle(5); grea->SetTitle( ↩
      "Real Values");

  TMultiGraph *mg = new TMultiGraph("toto", title.c_str());
  mg->Add(gest,"l3"); mg->Add(gobs,"P"); mg->Add(grea,"P");
  mg->Draw("A");
  mg->GetXaxis()->SetTitle("x_{1}");

  TLegend *leg= new TLegend(0.4,0.65,0.65,0.8);
  leg->AddEntry(gobs, "Observations", "p");
  leg->AddEntry(grea, "Real values", "p");
  leg->AddEntry(gest, "Estimated values", "lf");
  leg->Draw();
}


void modelerTestKriging()
{
  //Create dataserver and read training database
  TDataServer *tdsObs = new TDataServer("tdsObs","observations");
  tdsObs->fileDataRead("utf-1D-train.dat");
  const int nbObs = 6;

  // Canvas, divided in 2
  TCanvas *Can = new TCanvas("can","can",10,32,1600,700);
  TPad *apad = new TPad("apad","apad",0, 0.03, 1, 1);
  apad->Draw(); apad->Divide(2,1);

  //Name of the plot and correlation functions used
  string outplot="GaussAndMatern_1D_AllPoints.png";
  string Correl[2] = {"Gauss","Matern3/2"};

  //Pointer to needed objects, created in the loop
  TGPBuilder *gpb[2];
  TKriging *kg[2];
  TDataServer *tdsEstim[2];
  TLauncher2 *lkrig[2];

  stringstream sstr;
  //Looping over the two correlation function chosen
  for (int imet=0; imet<2; imet++)
  {
    //Create the TGPBuiler object with chosen option and find optimal parameters
    gpb[imet] = new TGPBuilder(tdsObs, "x1", "y", Correl[imet].c_str(), "");
    gpb[imet]->findOptimalParameters("LOO", 100, "Subplexe", 1000);

    //Get the kriging object
    kg[imet] = gpb[imet]->buildGP();

    //open the dataserver and read the testing basis
    sstr.str(""); sstr << "tdsEstim_" << imet;
    tdsEstim[imet] = new TDataServer(sstr.str().c_str(),"base de test");
    tdsEstim[imet]->fileDataRead("utf-1D-test.dat");

    //applied resulting kriging on test basis
    lkrig[imet] = new TLauncher2(tdsEstim[imet], kg[imet], "x1", "yEstim:vEstim");
    lkrig[imet]->solverLoop();

    //do the plot
    apad->cd(imet+1);
    PrintSolutions(tdsObs, tdsEstim[imet], Correl[imet]);

    stringstream sstr; TLatex *lat=new TLatex(); lat->SetNDC(); lat->SetTextSize(0.025);
    sstr.str(""); sstr << "RMSE (by Loo) " << kg[imet]->getLooRMSE();
    lat->DrawLatex(0.4,0.61,sstr.str().c_str());
    sstr.str(""); sstr << "Q2 " << kg[imet]->getLooQ2();
    lat->DrawLatex(0.4,0.57,sstr.str().c_str());

  }
}
```

The first function of this macro (called `PrintSolutions`) is a complex part that will not be detailed, used to represent the results.

The macro itself starts by reading the training database and storing it in a dataserver. A `TGPBuilder` is created with the chosen correlation function and the hyper-parameters are estimation by an optimisation procedure in:

```
gpb[imet] = new TGPBuilder(tdsObs, "x1", "y", Correl[imet].c_str(), "");
gpb[imet]->findOptimalParameters("LOO", 100, "Subplexe", 1000);

kg[imet] = gpb[imet]->buildGP()
```

The last line shows how to build and retrieve the newly created kriging object.

Finally, this kriging model is tested against the training database, thanks to a `TLauncher2` object, as following:

```
lkrig[imet] = new TLauncher2(tdsEstim[imet], kg[imet], "x1", "yEstim:vEstim");
lkrig[imet]->solverLoop();
```

XIV.6.14.3 Graph

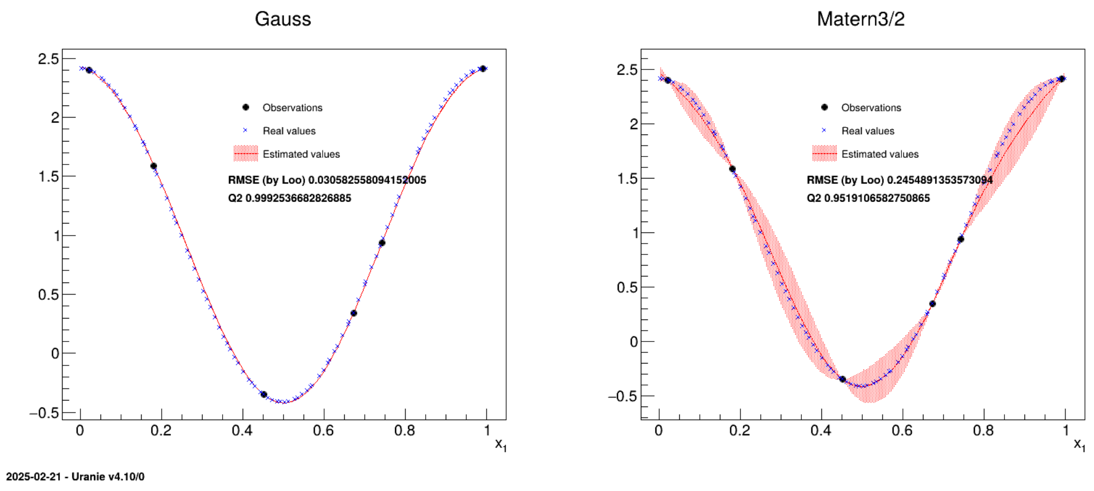


Figure XIV.72: Graph of the macro "`modelerTestKriging.C`"

XIV.7 Macros Optimizer

XIV.7.1 Macro "`optimizeFunctionRosenbrock.C`"

XIV.7.1.1 Objective

The objective of this macro is to perform a minimisation of a given function `rosenbrock` with the *Migrad* method.

XIV.7.1.2 Macro Uranie

```
{
  // Load the function
  gROOT->LoadMacro("UserFunctions.C");

  // Les variables de l'etude
  TAttribute *x = new TAttribute("x", -1.5, 1.5);
  x->setDefaultValue(-1.2);
  x->setStepValue(0.01);

  TAttribute *y = new TAttribute("y", -1.5,1.5);
  y->setDefaultValue(1.);
  y->setStepValue(0.01);

  TDataServer *tdsRosenbrock = new TDataServer("tdsRosenbrock", "Optimize Code externe ↩
      Rosenbrock via TDataServer");
  tdsRosenbrock->addAttribute(x);
  tdsRosenbrock->addAttribute(y);

  // Graph
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro optimizeFunctionRosenbrock" ↩
      ,5,64,1270,667);

  // Create an TOptimizer object from TDataServer and an anlystical function
  TOptimizer * topt = new TOptimizer(tdsRosenbrock, "rosenbrock","","out");
  // topt->setMethod(TOptimizer::kSimplex);
  // topt->setTolerance(1e-5);
  // topt->setPrintLevel(5);
  // topt->setMaxIterations(3);
  // topt->setMaxFunctionCalls(10);
  topt->optimize("same");
  tdsRosenbrock->exportData("_tds_rosenbrock_.dat");

}
```

The Rosenbrock's function is specified in the following file `UserFunctions.C` (the file can be found in `${URANIESYS}/s`
The `UserFunctions.C` is loaded to get the `rosenbrock` method:

```
gROOT->LoadMacro("UserFunctions.C");
```

The two `TAttribute` objects are the variables of the study: **x** and **y**. The intervals of definition and initial default values are set. The step value is set to *0.01* for the two variables:

```
TAttribute *x = new TAttribute("x", -1.5, 1.5);
x->setDefaultValue(-1.2);
x->setStepValue(0.01);

TAttribute *y = new TAttribute("y", -1.5,1.5);
y->setDefaultValue(1.);
y->setStepValue(0.01);
```

The `TOptimizer` object is initialised with the `TDataServer` containing both attributes, the name of the function to minimise, entry variables and output variable:

```
TOptimizer * topt = new TOptimizer(tdsRosenbrock, "rosenbronck", "", "out");
topt->optimize("same");
```

Results are exported in an ASCII file:

```
tdsRosenbrock->exportData("_tds_rosenbrock_.dat");
```

XIV.7.1.3 Graph

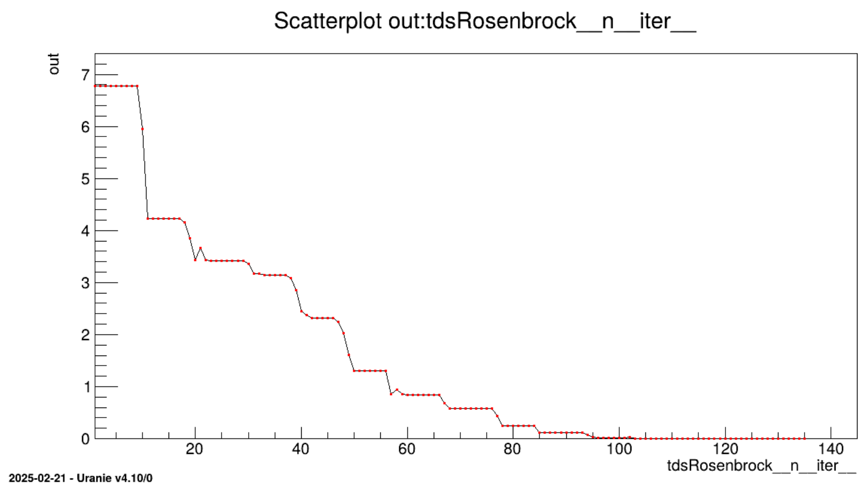


Figure XIV.73: Graph of the macro "`optimizeFunctionRosenbrock.C`"

XIV.7.2 Macro "`optimizeFunctionRosenbrockNewInputOutput.C`"

XIV.7.2.1 Objective

The objective of this macro is to perform a minimisation of a given function `rosenbrock` with the *Migrad* method. On top of the objective introduced in Section XIV.7.1.1, the idea is here to do the optimisation introducing new inputs (which are shifted values of the original ones) and add new output variables given simple mathematical formulae. Warning, this is slowing down the process as new values have to be computed on the fly, so we strongly recommend to change the loaded function instead. This functionnality has been introduced for consistency with the code optimisation.

XIV.7.2.2 Macro Uranie

```
{
  // Load the function
  gROOT->LoadMacro("UserFunctions.C");

  // Les variables de l'etude
  TAttribute *x = new TAttribute("x", -1.5, 1.5);
  x->setDefaultValue(-1.2);
  x->setStepValue(0.01);

  TAttribute *y = new TAttribute("y", -1.5,1.5);
  y->setDefaultValue(1.);
  y->setStepValue(0.01);

  TDataServer *tdsRosenbrock = new TDataServer("tdsRosenbrock", "Optimize Code externe ↩
      Rosenbrock via TDataServer");
  tdsRosenbrock->addAttribute(x);
  tdsRosenbrock->addAttribute(y);

  tdsRosenbrock->addAttribute("xshift","x-0.1");
  tdsRosenbrock->addAttribute("yshift","y+0.2");

  // Graph
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro ↩
      optimizeFunctionRosenbrockNewInputOutput",5,64,1270,667);

  // Create an TOptimizer object from TDataServer and an anlystical function
  TOptimizer * topt = new TOptimizer(tdsRosenbrock, "rosenbrock","xshift:yshift","out");
  topt->addOutputVariable("out+1:out*out:out*3");
  topt->selectCost("out+1");
  // topt->setMethod(TOptimizer::kSimplex);
  // topt->setTolerance(1e-5);
  // topt->setPrintLevel(5);
  // topt->setMaxIterations(3);
  // topt->setMaxFunctionCalls(10);
  topt->optimize("same");
  tdsRosenbrock->exportData("_tds_rosenbrock_.dat");

}
```

The main differences with Section XIV.7.1.2 are gathered in the next few blocks. It starts with the introduction of the new input attributes, defined through formulae:

```
tdsRosenbrock->addAttribute("xshift","x-0.1");
tdsRosenbrock->addAttribute("yshift","y+0.2");
```

The fact that the optimisation should be done on these newly defined variables is precised in the construction of the optimizer:

```
TOptimizer * topt = new TOptimizer(tdsRosenbrock, "rosenbronck", "xshift:yshift", "out");
```

The new output variables are defined through a dedicated method and the variable upon which the optimisation is performed is specified thanks to the `selectCost` function:

```
topt->addOutputVariable("out+1:out*out:out*3");
topt->selectCost("out+1");
```

The rest is consistent with what's done in Section XIV.7.1.2.

```
tdsRosenbrock->exportData("_tds_rosenbrock_.dat");
```

XIV.7.2.3 Graph

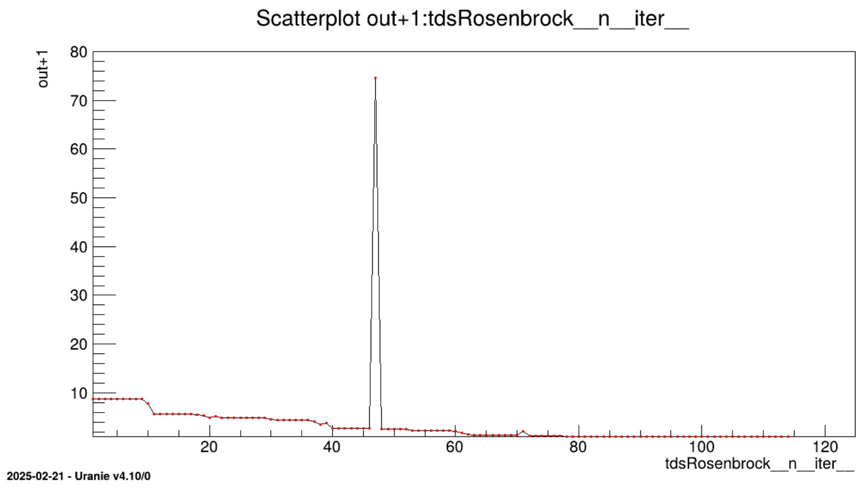


Figure XIV.74: Graph of the macro "optimizeFunctionRosenbrockNewInputOutput.C"

XIV.7.3 Macro "optimizeCodeRosenbrockKey.C"

XIV.7.3.1 Objective

The objective of this macro is to perform an optimisation of the `rosenbrock` function returned by the **rosenbrock** code (described in Section VII.2.1.3) with the values of the two attributes **x** and **y** read in an input file with "key=value" format, `input_rosenbrock_with_keys.dat`:

```
#
#
# inputfile for the \b rosenbrock code
# \date   mar jui 3 14:38:43 2007
# the two parameters
#

x  = -1.20 ;
y  = 1.0 ;
a = 10.0 ;
b = 1.0 ;
```

The output file, `_output_rosenbrock_with_keys_.dat`, is with "key=value" format and looks like:

```
X = -1.200000e+000 ;
Y = 1.000000e+000 ;
fval = 6.776000e+000 ;
fA = 1.000000e+001 ;
fB = 1.000000e+000 ;
```

where *fA* and *fB* are parameters of the `rosenbrock` function. *X* and *Y* are the values of attributes *x* and *y*, *fval* the cost variable.

XIV.7.3.2 Macro Uranie

```
{

  // The x attribute of the use case
  TAttribute *x = new TAttribute("x", -1.5, 1.5);
  x->setDefaultValue(-1.2);
  x->setStepValue(0.01);
  x->setFileKey("input_rosenbrock_with_keys.dat");

  // The y attribute of the use case
  TAttribute *y = new TAttribute("y", -1.5,1.5);
  y->setDefaultValue(1.);
  y->setStepValue(0.01);
  y->setFileKey("input_rosenbrock_with_keys.dat");

  // Define the DataServer and add the two attributes
  TDataServer *tdsRosenbrock = new TDataServer("tdsRosenbrock", "Optimize Code externe ↩
      Rosenbrock via TDataServer");
  tdsRosenbrock->addAttribute(x);
  tdsRosenbrock->addAttribute(y);

  // The output file of the code where values are stored in (key = value) format
  TOutputFileKey *fOutputFile = new TOutputFileKey("_output_rosenbrock_with_keys_.dat");
  fOutputFile->addAttribute(new TAttribute("fval"));

  // Create an TCode object for my code
  TCode *myRosenbrockCode = new TCode(tdsRosenbrock, "rosenbrock -k");
  // The working directory to launch the code
  //myRosenbrockCode->setWorkingDirectory(gSystem->Getenv("PWD") + TString("/ ↩
      tmpLanceurUranie/rosenbrock"));
  // Add the output file
  myRosenbrockCode->addOutputFile( fOutputFile );

   // Graph
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro optimizeCodeRosenbrockKey" ↩
      ,5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2,1);
  pad->cd(1);

  // Create an TOptimizer object from TDataServer and TCode objects
  TOptimizer * topt = new TOptimizer(tdsRosenbrock, myRosenbrockCode);
  topt->setMethod(TOptimizer::kSimplex);
  // topt->setTolerance(1e-5);
  // topt->setPrintLevel(5);
  // topt->setMaxIterations(3);
  // topt->setMaxFunctionCalls(10);
  topt->optimize("same");
  tdsRosenbrock->exportData("_tds_rosenbrock_.dat");


  pad->cd(2);
  TF2 *frosenbrok = new TF2("fcnRosenbrock","(y-x*x)*(y-x*x)* [0] + ( 1.0 -x ) * ( 1.0 -x ) ↩
      *[1]",-2.0, 2.0, -2.0,2.0);
  frosenbrok->SetParameter(0,10.0);
  frosenbrok->SetParameter(1,1.0);
  frosenbrok->Draw("cont1z");
  tdsRosenbrock->draw("y:x", "", "same1");

  tdsRosenbrock->getTuple()->SetMarkerColor(4);
  tdsRosenbrock->getTuple()->SetMarkerStyle(8);
  tdsRosenbrock->getTuple()->SetMarkerSize(.90);
  tdsRosenbrock->Draw("y:x", Form("%s==1", tdsRosenbrock->getIteratorName()), "psame");
  tdsRosenbrock->getTuple()->SetMarkerColor(50);
  tdsRosenbrock->Draw("y:x", Form("%s==%d", tdsRosenbrock->getIteratorName(), tdsRosenbrock ↩
      ->getNPatterns()), "psame");

}
```

The `TAttribute` objects **x** and **y** are linked to the input file `input_rosenbrock_with_keys.dat`:

```
TAttribute *x = new TAttribute("x", -1.5, 1.5);
x->setDefaultValue(-1.2);
x->setStepValue(0.01);
x->setFileKey("input_rosenbrock_with_keys.dat");

TAttribute *y = new TAttribute("y", -1.5,1.5);
y->setDefaultValue(1.);
y->setStepValue(0.01);
y->setFileKey("input_rosenbrock_with_keys.dat");
```

Instantiating the output file:

```
TOutputFileKey *fOutputFile = new TOutputFileKey("_output_rosenbrock_with_keys_.dat");
```

The cost variable is added to the output file as a new `TAttribute`:

```
fOutputFile->addAttribute(new TAttribute("fval"));
```

Instantiating the `TCode` object, the **rosenbrock** code is launched with the *-k* option. The input file searched by the code will then be with type "key=value":

```
TCode *myRosenbrockCode = new TCode(tdsRosenbrock, "rosenbrock -v -k");
```

The `TOptimizer` object is initialised with the `TDataServer` containing data and the `TCode` object. The optimisation is built with the *Simplex* method:

```
TOptimizer * topt = new TOptimizer(tdsRosenbrock, myRosenbrockCode);
topt->setMethod(TOptimizer::kSimplex);
topt->optimize("same");
```

The `TDataServer` is exported in an ASCII file:

```
tdsRosenbrock->exportData("_tds_rosenbrock_.dat");
```

XIV.7.3.3 Graph

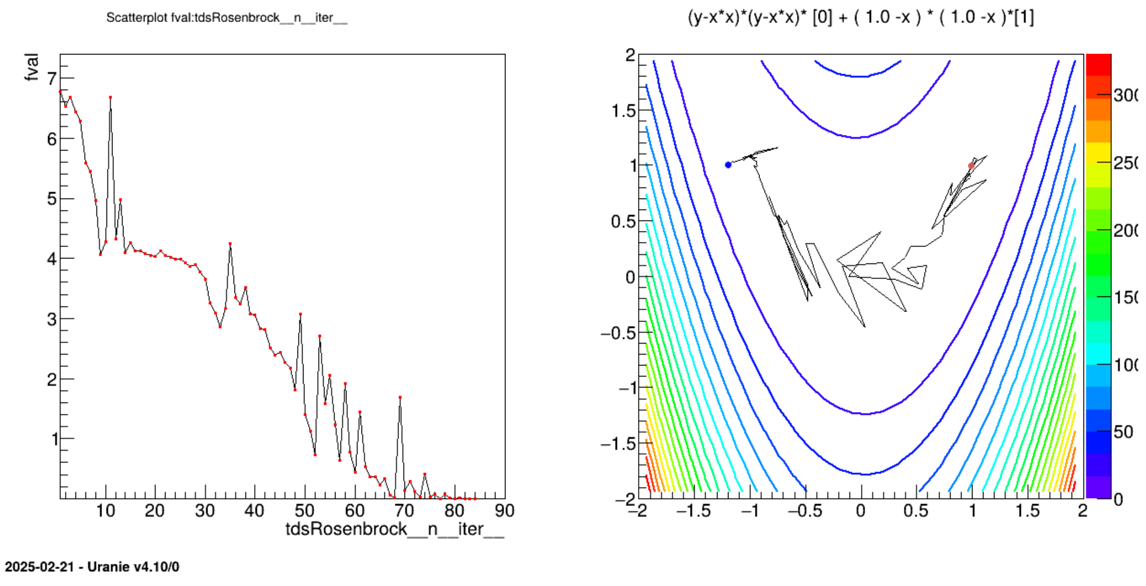


Figure XIV.75: Graph of the macro "optimizeCodeRosenbrockKey.C"

XIV.7.3.4 Console

```
Processing optimizeCodeRosenbrockKey.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 **   sLibraryName[Minuit2]
 **    sMethodName[Simplex]
 ** Problem[kMinimizeCode]
 ** input :: ivar[0/3] name[x]
 ** input :: ivar[1/3] name[y]
 ** output :: ivar[2/3] name[fval]
 ** TMultiGenCode::init _sCost[fval] _nCost[ 1]
 ** _sCost[fval]
 ********** Print state [0] option[Init]
 **   name [ x] Origin[kAttribute] Value[-1.2]
 **   name [ y] Origin[kAttribute] Value[1]
 **   name [ fval] Origin[kAttribute] Value[1.23457]
 ********** End Of Print state [0]
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${SOURCEDIR}/launCHER/souRCE/TCode.cxx] Line[1350]
 <URANIE::INFO> TCode::createDirectory Method
 <URANIE::INFO>  The directory "${RUNNINGDIR}/URANIE/UranieLauncher_1" does not exist
 <URANIE::INFO>  URANIE creates it for you
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>
 ********** Print state [84] option[Final]
 **   name [ x] Origin[kAttribute] Value[0.995289]
 **   name [ y] Origin[kAttribute] Value[0.99263]
 **   name [ fval] Origin[kAttribute] Value[6.34168e-05]
 ********** End Of Print state [84]
```

XIV.7.4 Macro "`optimizeCodeRosenbrockKeyNewInputOutput.C`"

XIV.7.4.1 Objective

The objectives of this macro are the same as the ones detailed in Section XIV.7.3.1. On top of these, as for Section XIV.7.2.1, the idea is here to do the optimisation introducing new inputs (which are shifted values of the original ones) and add new output variables given simple mathematical formulae.

XIV.7.4.2 Macro Uranie

```
{

  // The x attribute of the use case
  TAttribute *x = new TAttribute("x", -1.5, 1.5);
  x->setDefaultValue(-1.2);
  x->setStepValue(0.01);

  // The y attribute of the use case
  TAttribute *y = new TAttribute("y", -1.5,1.5);
  y->setDefaultValue(1.);
  y->setStepValue(0.01);

  // Define the DataServer and add the two attributes
  TDataServer *tdsRosenbrock = new TDataServer("tdsRosenbrock", "Optimize Code externe ↩
      Rosenbrock via TDataServer");
  tdsRosenbrock->addAttribute(x);
  tdsRosenbrock->addAttribute(y);

  tdsRosenbrock->addAttribute("xshift","x-0.1");
  tdsRosenbrock->getAttribute("xshift")->setFileKey("input_rosenbrock_with_keys.dat","x");
  tdsRosenbrock->addAttribute("yshift","y+0.2");
  tdsRosenbrock->getAttribute("yshift")->setFileKey("input_rosenbrock_with_keys.dat","y");


  // The output file of the code where values are stored in (key = value) format
  TOutputFileKey *fOutputFile = new TOutputFileKey("_output_rosenbrock_with_keys_.dat");
  fOutputFile->addAttribute(new TAttribute("fval"));

  // Create an TCode object for my code
  TCode *myRosenbrockCode = new TCode(tdsRosenbrock, "rosenbrock -k");
  // The working directory to launch the code
  //myRosenbrockCode->setWorkingDirectory(gSystem->Getenv("PWD") + TString("/ ↩
      tmpLanceurUranie/rosenbrock"));
  // Add the output file
  myRosenbrockCode->addOutputFile( fOutputFile );

   // Graph
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro  ↩
      optimizeCodeRosenbrockKeyNewInputOutput",5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2,1);
  pad->cd(1);

  // Create an TOptimizer object from TDataServer and TCode objects
  TOptimizer * topt = new TOptimizer(tdsRosenbrock, myRosenbrockCode);
  topt->setMethod(TOptimizer::kSimplex);
  topt->addOutputVariable("fval+1:fval*fval:fval*3");
  topt->selectCost("fval+1");
  topt->setTolerance(1e-3);
  // topt->setPrintLevel(5);
  // topt->setMaxIterations(3);
  // topt->setMaxFunctionCalls(10);

  topt->optimize("same");
  tdsRosenbrock->exportData("_tds_rosenbrock_.dat");


  pad->cd(2);
  TF2 * frosenbrok = new TF2("fcnRosenbrock","(y-x*x)*(y-x*x)* [0] + ( 1.0 -x ) * ( 1.0 -x  ↩
      )*[1]",-2.0, 2.0, -2.0,2.0);
  frosenbrok->SetParameter(0,10.0);
  frosenbrok->SetParameter(1,1.0);
  frosenbrok->Draw("cont1z");
  tdsRosenbrock->draw("y:x", "", "samel");

  tdsRosenbrock->getTuple()->SetMarkerColor(4);
  tdsRosenbrock->getTuple()->SetMarkerStyle(8);
  tdsRosenbrock->getTuple()->SetMarkerSize(.90);
  tdsRosenbrock->Draw("y:x", Form("%s==1", tdsRosenbrock->getIteratorName()), "psame");
  tdsRosenbrock->getTuple()->SetMarkerColor(50);
  tdsRosenbrock->Draw("y:x", Form("%s==%d", tdsRosenbrock->getIteratorName(), tdsRosenbrock ↩
      ->getNPatterns()), "psame");

}
```

The main differences with Section XIV.7.4.2 are gathered in the next few blocks. It starts with the introduction of the new input attributes, defined through formulae. The new attributes are connected to the input file thanks to the `setFileKey` (and these methods are not called from the original attributes anymore):

```
tdsRosenbrock->addAttribute("xshift","x-0.1");
tdsRosenbrock->getAttribute("xshift")->setFileKey("input_rosenbrock_with_keys.dat","x");
tdsRosenbrock->addAttribute("yshift","y+0.2");
tdsRosenbrock->getAttribute("yshift")->setFileKey("input_rosenbrock_with_keys.dat","y");
```

Once settled, the remaining modifications to be done with respect to Section XIV.7.3.2 are the creation of the new output variables and the change of variable to be used as cost function.

```
topt->addOutputVariable("fval+1:fval*fval:fval*3");
topt->selectCost("fval+1");
```

This, as for the use-case discussed in Section XIV.7.2, will lead to new results as shown in Section XIV.7.4.4: the minimisation is performed on the two new inputs, and the minimimun is found when these values are close to one. Looking at the original variables, the value toward which they converge is 1.1 and 0.8 respectively for x and y.

XIV.7.4.3 Graph

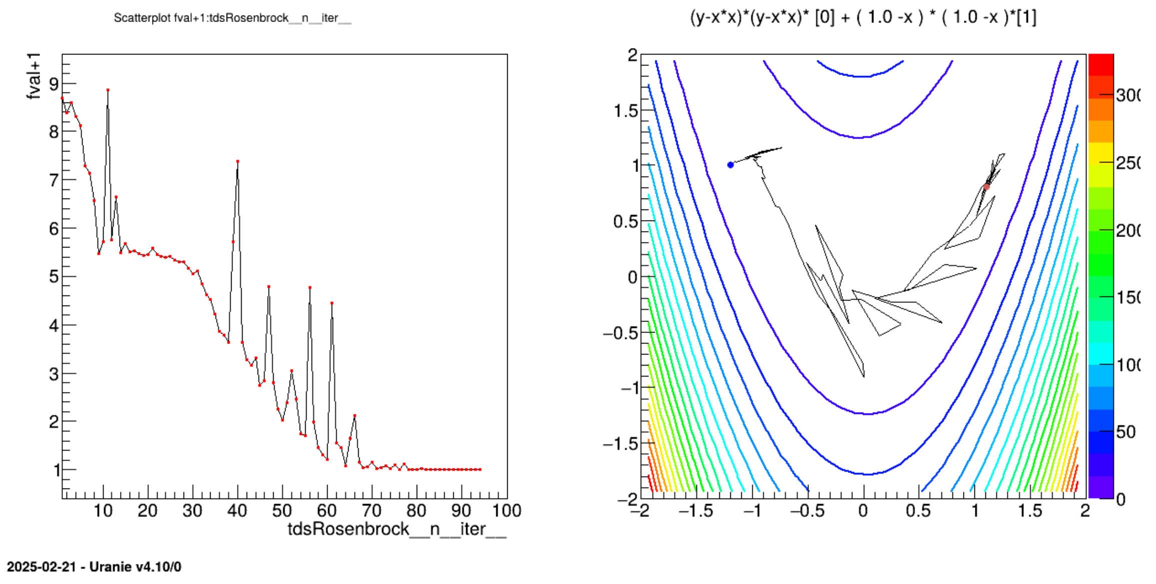


Figure XIV.76: Graph of the macro "optimizeCodeRosenbrockKeyNewInputOutput.C"

XIV.7.4.4 Console

```
Processing optimizeCodeRosenbrockKeyNewInputOutput.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 **   sLibraryName[Minuit2]
 **    sMethodName[Simplex]
 ** Problem[kMinimizeCode]
 ** input :: ivar[0/8] name[x]
 ** input :: ivar[1/8] name[y]
 ** input :: ivar[2/8] name[xshift]
 ** input :: ivar[3/8] name[yshift]
 ** output :: ivar[4/8] name[fval]
 ** output :: ivar[5/8] name[fval+1]
 ** output :: ivar[6/8] name[fval*fval]
 ** output :: ivar[7/8] name[fval*3]
 ** TMultiGenCode::init _sCost[fval+1] _nCost[ 2]
 ** TMultiGenCode::init Creating the formula for new inputs
 ** _sCost[fval+1]
 ********** Print state [0] option[Init]
 **   name [ x] Origin[kAttribute] Value[-1.2]
 **   name [ y] Origin[kAttribute] Value[1]
 **   name [ fval+1] Origin[kAttribute] Value[1.23457]
 ********** End Of Print state [0]
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${SOURCEDIR}/launCHER/souRCE/TCode.cxx] Line[1350]
 <URANIE::INFO> TCode::createDirectory Method
 <URANIE::INFO>  The directory "${RUNNINGDIR}/URANIE/UranieLauncher_1" does not exist
```

```
<URANIE::INFO>  URANIE creates it for you
<URANIE::INFO> *** END of URANIE INFORMATION ***
<URANIE::INFO>
********** Print state [94] option[Final]
**   name [ x] Origin[kAttribute] Value[1.10427]
**   name [ y] Origin[kAttribute] Value[0.808468]
**   name [ fval+1] Origin[kAttribute] Value[1.00002]
********** End Of Print state [94]
```

XIV.7.5 Macro "`optimizeCodeRosenbrockRow.C`"

XIV.7.5.1 Objective

The objective of this macro is to perform optimisation of the `rosenbrock` function returned by the **rosenbrock** code (described in Section VII.2.1.3) with the values of the two attributes **x** and **y** read in an input file with "key=value" format, `input_rosenbrock_with_keys.dat`:

```
#
#
# inputfile for the \b rosenbrock code
# \date   mar jui 3 14:38:43 2007
# the two parameters
#

x   = -1.20 ;
y   = 1.0 ;
a = 10.0 ;
b = 1.0 ;
```

The output file, `_output_rosenbrock_with_values_rows_.dat`, is with "values in rows" format and looks like

```
#COLUMN_NAMES: x | y | fval | pA | pB

-1.200000e+000  1.000000e+000 6.776000e+000 1.000000e+001 1.000000e+000
```

where *pA* and *pB* are parameters of the `rosenbrock` function. *X* and *Y* are the values of attributes *x* and *y*, *fval* the cost variable.

XIV.7.5.2 Macro Uranie

```
{

  // The x attribute of the use case
  TAttribute *x = new TAttribute("x", -1.5, 1.5);
  x->setDefaultValue(-1.2);
  x->setStepValue(0.01);
  x->setFileKey("input_rosenbrock_with_keys.dat");

  // The y attribute of the use case
  TAttribute *y = new TAttribute("y", -1.5,1.5);
  y->setDefaultValue(1.);
  y->setStepValue(0.01);
  y->setFileKey("input_rosenbrock_with_keys.dat");

  // Define the DataServer and add the two attributes
  TDataServer *tdsRosenbrock = new TDataServer("tdsRosenbrock", "Optimize Code externe  ↩
      Rosenbrock via TDataServer");
  tdsRosenbrock->addAttribute(x);
  tdsRosenbrock->addAttribute(y);

  // The output file of the code where values are in row
  TOutputFileRow *fOutputFile = new TOutputFileRow("_output_rosenbrock_with_values_rows_. ↩
      dat");
  fOutputFile->addAttribute(new TAttribute("fval"), 3);

  // Create an TCode object for my code
  TCode *myRosenbrockCode = new TCode(tdsRosenbrock, "rosenbrock -k");
  // The working directory to launch the code
  //myRosenbrockCode->setWorkingDirectory(gSystem->Getenv("PWD") + TString("/ ↩
      tmpLanceurUranie/rosenbrock"));
  // Add the output file
  myRosenbrockCode->addOutputFile( fOutputFile );

  // Graph
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro optimizeCodeRosenbrockRow" ↩
      ,5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2,1);
  pad->cd(1);

  // Create an TOptimizer object from TDataServer and TCode objects
  TOptimizer * topt = new TOptimizer(tdsRosenbrock, myRosenbrockCode);
  topt->setMethod(TOptimizer::kSimplex);
  // topt->setTolerance(1e-5);
  // topt->setPrintLevel(5);
  // topt->setMaxIterations(3);
  // topt->setMaxFunctionCalls(10);

  topt->optimize("same");
  tdsRosenbrock->exportData("_tds_rosenbrock_.dat");


  pad->cd(2);
  TF2 * frosenbrok = new TF2("fcnRosenbrock","(y-x*x)*(y-x*x)* [0] + ( 1.0 -x ) * ( 1.0 -x  ↩
      )*[1]",-2.0, 2.0, -2.0,2.0);
  frosenbrok->SetParameter(0,10.0);
  frosenbrok->SetParameter(1,1.0);
  frosenbrok->Draw("cont1z");
  tdsRosenbrock->draw("y:x", "", "samel");

  tdsRosenbrock->getTuple()->SetMarkerColor(4);
  tdsRosenbrock->getTuple()->SetMarkerStyle(8);
  tdsRosenbrock->getTuple()->SetMarkerSize(.90);
  tdsRosenbrock->Draw("y:x", Form("%s==1", tdsRosenbrock->getIteratorName()), "psame");
  tdsRosenbrock->getTuple()->SetMarkerColor(50);
  tdsRosenbrock->Draw("y:x", Form("%s==%d", tdsRosenbrock->getIteratorName(), tdsRosenbrock ↩
      ->getNPatterns()), "psame");

}
```

The `TAttribute` objects **x** and **y** are linked to the input file `input_rosenbrock_with_keys.dat`:

```
TAttribute *x = new TAttribute("x", -1.5, 1.5);
x->setDefaultValue(-1.2);
x->setStepValue(0.01);
x->setFileKey("input_rosenbrock_with_keys.dat");
```

```
TAttribute *y = new TAttribute("y", -1.5,1.5);
y->setDefaultValue(1.);
y->setStepValue(0.01);
y->setFileKey("input_rosenbrock_with_keys.dat");
```

Instantiating the output file:

```
TOutputFileRow *fOutputFile = new TOutputFileRow("_output_rosenbrock_with_values_rows_.dat" ↩
    );
```

The cost variable is added to the output file as a new `TAttribute`:

```
fOutputFile->addAttribute(new TAttribute("fval"));
```

Instantiating the `TCode` object, the **rosenbrock** code is launched with the *-k* option. The input file searched by the code will then be with type "key=value":

```
TCode *myRosenbrockCode = new TCode(tdsRosenbrock, "rosenbrock -v -k");
```

The `TOptimizer` object is initialised with the `TDataServer` containing data and the `TCode` object. The optimisation is built with the *Simplex* method:

```
TOptimizer * topt = new TOptimizer(tdsRosenbrock, myRosenbrockCode);
topt->setMethod(TOptimizer::kSimplex);
topt->optimize("same");
```

The `TDataServer` is exported in an ASCII file:

```
tdsRosenbrock->exportData("_tds_rosenbrock_.dat");
```

XIV.7.5.3 Graph

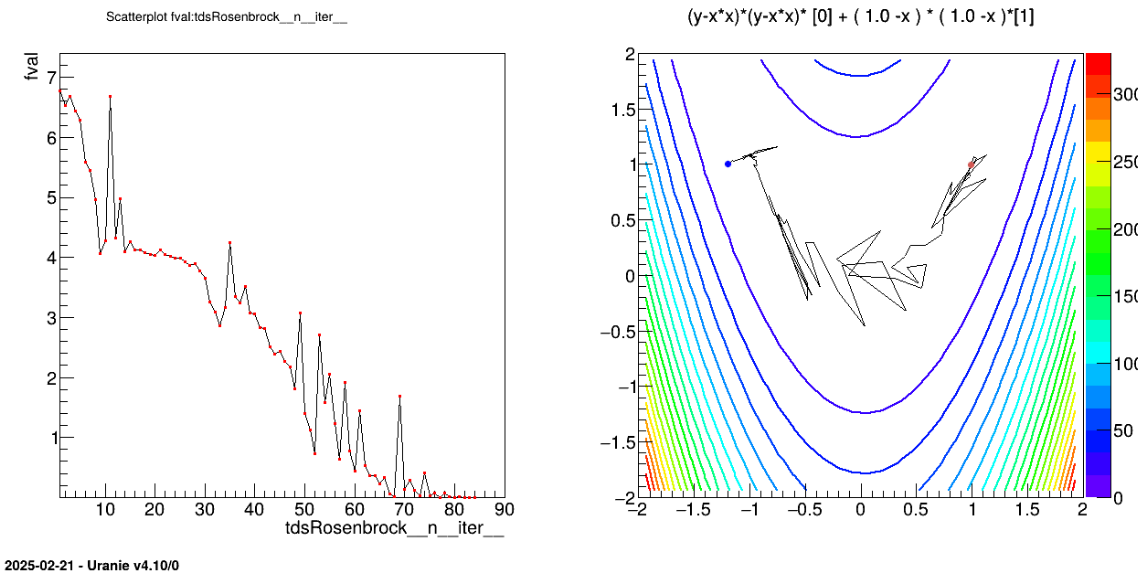


Figure XIV.77: Graph of the macro "optimizeCodeRosenbrockRow.C"

XIV.7.5.4 Console

```
Processing optimizeCodeRosenbrockRow.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 **   sLibraryName[Minuit2]
 **    sMethodName[Simplex]
 ** Problem[kMinimizeCode]
 ** input :: ivar[0/3] name[x]
 ** input :: ivar[1/3] name[y]
 ** output :: ivar[2/3] name[fval]
 ** TMultiGenCode::init _sCost[fval] _nCost[ 1]
 ** _sCost[fval]
 ********** Print state [0] option[Init]
 **   name [ x] Origin[kAttribute] Value[-1.2]
 **   name [ y] Origin[kAttribute] Value[1]
 **   name [ fval] Origin[kAttribute] Value[1.23457]
 ********** End Of Print state [0]
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${SOURCEDIR}/launCHER/souRCE/TCode.cxx] Line[1350]
 <URANIE::INFO> TCode::createDirectory Method
 <URANIE::INFO>  The directory "${RUNNINGDIR}/URANIE/UranieLauncher_1" does not exist
 <URANIE::INFO>  URANIE creates it for you
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>
 ********** Print state [84] option[Final]
 **   name [ x] Origin[kAttribute] Value[0.995289]
 **   name [ y] Origin[kAttribute] Value[0.99263]
 **   name [ fval] Origin[kAttribute] Value[6.34168e-05]
 ********** End Of Print state [84]
```

XIV.7.6 Macro "`optimizeCodeRosenbrockKeyRowRecreate.C`"

XIV.7.6.1 Objective

The objective of this macro is to perform optimisation of the `rosenbrock` function returned by the **rosenbrock** code (described in Section VII.2.1.3) with the values of the two attributes **x** and **y** each defined in two input files, one with "key=value" format, `input_rosenbrock_with_keys.dat`:

```
#
#
# inputfile for the \b rosenbrock code
# \date   mar jui 3 14:38:43 2007
# the two parameters
#

x  = -1.20 ;
y  = 1.0 ;
a = 10.0 ;
b = 1.0 ;
```

the other with "values in rows" format, `input_rosenbrock_with_values_rows.dat`:

```
-1.20 1.0
```

Only the option given to the **rosenbrock** code will determine the input file effectively used. The output file, `_output_rosenbrock_with_keys_.dat`, is with "key=value" format and looks like:

```
X = -1.200000e+000 ;
Y = 1.000000e+000 ;
fval = 6.776000e+000 ;
fA = 1.000000e+001 ;
fB = 1.000000e+000 ;
```

where *fA* and *fB* are parameters of the `rosenbrock` function. *X* and *Y* are the values of attributes *x* and *y*, *fval* the cost variable.

XIV.7.6.2 Macro Uranie

```
{

  // The x attribute of the use case
  TAttribute *x = new TAttribute("x", -1.5, 1.5);
  x->setDefaultValue(-1.2);
  x->setStepValue(0.01);
  x->setFileKey("input_rosenbrock_with_keys.dat");
  x->setFileKey("input_rosenbrock_with_values_rows.dat","x","%e",TAttributeFileKey::kNewRow ↩
      );

  // The y attribute of the use case
  TAttribute *y = new TAttribute("y", -1.5,1.5);
  y->setDefaultValue(1.);
  y->setStepValue(0.01);
  y->setFileKey("input_rosenbrock_with_keys.dat");
  y->setFileKey("input_rosenbrock_with_values_rows.dat","y","%e",TAttributeFileKey::kNewRow ↩
      );

  // Define the DataServer and add the two attributes
  TDataServer *tdsRosenbrock = new TDataServer("tdsRosenbrock", "Optimize Code externe  ↩
      Rosenbrock via TDataServer");
  tdsRosenbrock->addAttribute(x);
  tdsRosenbrock->addAttribute(y);

  // The output file of the code where values are stored in (key = value) format
  TOutputFileKey *fOutputFile = new TOutputFileKey("_output_rosenbrock_with_keys_.dat");
  fOutputFile->addAttribute(new TAttribute("fval"));

  // Create an TCode object for my code
  TCode *myRosenbrockCode = new TCode(tdsRosenbrock, "rosenbrock -r");
  // Add the output file
  myRosenbrockCode->addOutputFile( fOutputFile );

  // Graph
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro  ↩
      optimizeCodeRosenbrockKeyRowRecreate",5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2,1);
  pad->cd(1);

  // Create an TOptimizer object from TDataServer and TCode objects
  TOptimizer * topt = new TOptimizer(tdsRosenbrock, myRosenbrockCode);
  topt->setMethod(TOptimizer::kSimplex);
  //   topt->setTolerance(1e-5);
  //   topt->setPrintLevel(5);
  //   topt->setMaxIterations(3);
  //   topt->setMaxFunctionCalls(10);

  topt->optimize("same");
  tdsRosenbrock->exportData("_tds_rosenbrock_.dat");


  pad->cd(2);
  TF2 * frosenbrok = new TF2("fcnRosenbrock","(y-x*x)*(y-x*x)* [0] + ( 1.0 -x ) * ( 1.0 -x  ↩
      )*[1]",-2.0, 2.0, -2.0,2.0);
  frosenbrok->SetParameter(0,10.0);
  frosenbrok->SetParameter(1,1.0);
  frosenbrok->Draw("cont1z");
  tdsRosenbrock->draw("y:x", "", "same1");

  tdsRosenbrock->getTuple()->SetMarkerColor(4);
  tdsRosenbrock->getTuple()->SetMarkerStyle(8);
  tdsRosenbrock->getTuple()->SetMarkerSize(.90);
  tdsRosenbrock->Draw("y:x", Form("%s==1", tdsRosenbrock->getIteratorName()), "psame");
  tdsRosenbrock->getTuple()->SetMarkerColor(50);
  tdsRosenbrock->Draw("y:x", Form("%s==%d", tdsRosenbrock->getIteratorName(), tdsRosenbrock ↩
      ->getNPatterns()), "psame");

}
```

The `TAttribute` objects **x** and **y** are linked to two input files `input_rosenbrock_with_keys.dat` with "key=value" format and `input_rosenbrock_with_values_rows.dat` with "values in rows" format:

```
TAttribute *x = new TAttribute("x", -1.5, 1.5);
x->setDefaultValue(-1.2);
x->setStepValue(0.01);
x->setFileKey("input_rosenbrock_with_keys.dat");
x->setFileKey("input_rosenbrock_with_values_rows.dat","x","%e",TAttributeFileKey::kNewRow);

TAttribute *y = new TAttribute("y", -1.5,1.5);
y->setDefaultValue(1.);
y->setStepValue(0.01);
y->setFileKey("input_rosenbrock_with_keys.dat");
y->setFileKey("input_rosenbrock_with_values_rows.dat","y","%e",TAttributeFileKey::kNewRow);
```

Instantiating the output file:

```
TOutputFileKey *fOutputFile = new TOutputFileKey("_output_rosenbrock_with_keys_.dat");
```

The cost variable is added to the output file as a new `TAttribute`:

```
fOutputFile->addAttribute(new TAttribute("fval"));
```

Instantiating the `TCode` object, the **rosenbrock** code is launched with the *-k* option. The input file searched by the code will then be with type "key=value":

```
TCode *myRosenbrockCode = new TCode(tdsRosenbrock, "rosenbrock -v -k");
```

The `TOptimizer` object is initialised with the `TDataServer` containing data and the `TCode` object. The optimisation is built with the *Simplex* method:

```
TOptimizer * topt = new TOptimizer(tdsRosenbrock, myRosenbrockCode);
topt->setMethod(TOptimizer::kSimplex);
topt->optimize("same");
```

The `TDataServer` is exported in an ASCII file:

```
tdsRosenbrock->exportData("_tds_rosenbrock_.dat");
```

XIV.7.6.3 Graph

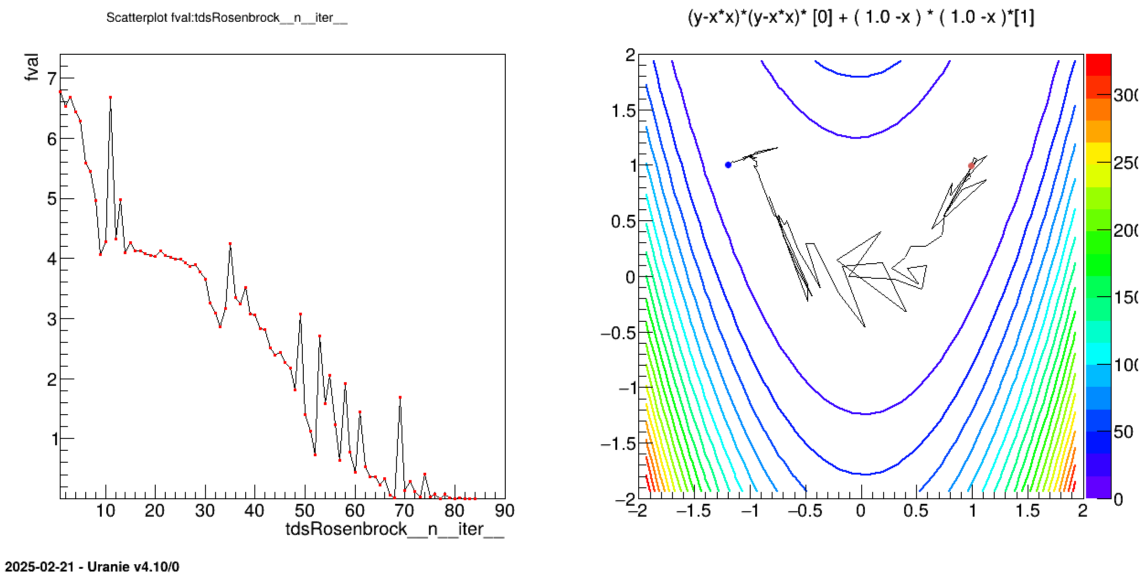


Figure XIV.78: Graph of the macro "optimizeCodeRosenbrockKeyRowRecreate.C"

XIV.7.6.4 Console

```
Processing optimizeCodeRosenbrockKeyRowRecreate.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 **   sLibraryName[Minuit2]
 **    sMethodName[Simplex]
 ** Problem[kMinimizeCode]
 ** input :: ivar[0/3] name[x]
 ** input :: ivar[1/3] name[y]
 ** output :: ivar[2/3] name[fval]
 ** TMultiGenCode::init _sCost[fval] _nCost[ 1]
 ** _sCost[fval]
 ********** Print state [0] option[Init]
 **   name [ x] Origin[kAttribute] Value[-1.2]
 **   name [ y] Origin[kAttribute] Value[1]
 **   name [ fval] Origin[kAttribute] Value[1.23457]
 ********** End Of Print state [0]
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${SOURCEDIR}/launCHER/souRCE/TCode.cxx] Line[1350]
 <URANIE::INFO> TCode::createDirectory Method
 <URANIE::INFO>  The directory "${RUNNINGDIR}/URANIE/UranieLauncher_1" does not exist
 <URANIE::INFO>  URANIE creates it for you
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>
 ********** Print state [84] option[Final]
 **   name [ x] Origin[kAttribute] Value[0.995289]
 **   name [ y] Origin[kAttribute] Value[0.99263]
 **   name [ fval] Origin[kAttribute] Value[6.34168e-05]
 ********** End Of Print state [84]
```

XIV.7.7 Macro "optimizeCodeRosenbrockRowRecreate.C"

XIV.7.7.1 Objective

The objective of this macro is to perform optimisation of the `rosenbrock` function returned by the **rosenbrock** code (described in Section VII.2.1.3) with the values of the two attributes **x** and **y** read in an input file with "values in rows" format, `input_rosenbrock_with_values_rows.dat`, written on the fly:

```
-1.20 1.0
```

The output file, `_output_rosenbrock_with_keys_.dat`, is with "key=value" format and looks like:

```
X = -1.200000e+000 ;
Y = 1.000000e+000 ;
fval = 6.776000e+000 ;
fA = 1.000000e+001 ;
fB = 1.000000e+000 ;
```

where *fA* and *fB* are parameters of the `rosenbrock` function. *X* and *Y* are the values of attributes *x* and *y*, *fval* the cost variable.

XIV.7.7.2 Macro Uranie

```
{

    // The x attribute of the use case
  TAttribute *x = new TAttribute("x", -1.5, 1.5);
  x->setDefaultValue(-1.2);
  x->setStepValue(0.01);
  x->setFileKey("input_rosenbrock_with_values_rows.dat","x","%e",TAttributeFileKey::kNewRow ↩
      );

  // The y attribute of the use case
  TAttribute *y = new TAttribute("y", -1.5,1.5);
  y->setDefaultValue(1.);
  y->setStepValue(0.01);
  y->setFileKey("input_rosenbrock_with_values_rows.dat","y","%e",TAttributeFileKey::kNewRow ↩
      );

  // Define the DataServer and add the two attributes
  TDataServer *tdsRosenbrock = new TDataServer("tdsRosenbrock", "Optimize Code externe  ↩
      Rosenbrock via TDataServer");
  tdsRosenbrock->addAttribute(x);
  tdsRosenbrock->addAttribute(y);


  // The output file of the code where values are stored in (key = value) format
  TOutputFileKey *fOutputFile = new TOutputFileKey("_output_rosenbrock_with_keys_.dat");
  fOutputFile->addAttribute(new TAttribute("fval"));

  // Create an TCode object for my code
  TCode *myRosenbrockCode = new TCode(tdsRosenbrock, "rosenbrock -r");
  // The working directory to launch the code
  //myRosenbrockCode->setWorkingDirectory(gSystem->Getenv("PWD") + TString("/ ↩
      tmpLanceurUranie/rosenbrock"));
  // Add the output file
  myRosenbrockCode->addOutputFile( fOutputFile );

  // Graph
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro  ↩
      optimizeCodeRosenbrockRowRecreate",5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2,1);
  pad->cd(1);

  // Create an TOptimizer object from TDataServer and TCode objects
  TOptimizer * topt = new TOptimizer(tdsRosenbrock, myRosenbrockCode);
  topt->setMethod(TOptimizer::kSimplex);
  //   topt->setTolerance(1e-5);
  //   topt->setPrintLevel(5);
  //   topt->setMaxIterations(3);
  //   topt->setMaxFunctionCalls(10);

  topt->optimize();
  tdsRosenbrock->exportData("_tds_rosenbrock_.dat");

  pad->cd(2);
  TF2 * frosenbrok = new TF2("fcnRosenbrock","(y-x*x)*(y-x*x)* [0] + ( 1.0 -x ) * ( 1.0 -x  ↩
      )*[1]",-2.0, 2.0, -2.0,2.0);
  frosenbrok->SetParameter(0,10.0);
  frosenbrok->SetParameter(1,1.0);
  frosenbrok->Draw("cont1z");
  tdsRosenbrock->draw("y:x", "", "same1");

  tdsRosenbrock->getTuple()->SetMarkerColor(4);
  tdsRosenbrock->getTuple()->SetMarkerStyle(8);
  tdsRosenbrock->getTuple()->SetMarkerSize(.90);
  tdsRosenbrock->Draw("y:x", Form("%s==1", tdsRosenbrock->getIteratorName()), "psame");
  tdsRosenbrock->getTuple()->SetMarkerColor(50);
  tdsRosenbrock->Draw("y:x", Form("%s==%d", tdsRosenbrock->getIteratorName(), tdsRosenbrock ↩
      ->getNPatterns()), "psame");

}
```

The TAttribute objects **x** and **y** are linked to the input file input_rosenbrock_with_values_rows.dat:

```
TAttribute *x = new TAttribute("x", -1.5, 1.5);
x->setDefaultValue(-1.2);
x->setStepValue(0.01);
x->setFileKey("input_rosenbrock_with_values_rows.dat","x","%e",TAttributeFileKey::kNewRow);

TAttribute *y = new TAttribute("y", -1.5,1.5);
y->setDefaultValue(1.);
y->setStepValue(0.01);
y->setFileKey("input_rosenbrock_with_values_rows.dat","y","%e",TAttributeFileKey::kNewRow);
```

Instantiating the output file:

```
TOutputFileKey *fOutputFile = new TOutputFileKey("_output_rosenbrock_with_keys_.dat");
```

The cost variable is added to the output file as a new TAttribute:

```
fOutputFile->addAttribute(new TAttribute("fval"));
```

Instantiating the TCode object, the **rosenbrock** code is launched with the *-r* option. The input file searched by the code will then be with type "values in rows":

```
TCode *myRosenbrockCode = new TCode(tdsRosenbrock, "rosenbrock -v -r");
```

The TOptimizer object is initialised with the TDataServer containing data and the TCode object. The optimisation is built with the *Simplex* method:

```
TOptimizer * topt = new TOptimizer(tdsRosenbrock, myRosenbrockCode);
topt->setMethod(TOptimizer::kSimplex);
topt->optimize("same");
```

The TDataServer is exported in an ASCII file:

```
tdsRosenbrock->exportData("_tds_rosenbrock_.dat");
```

XIV.7.7.3 Graph

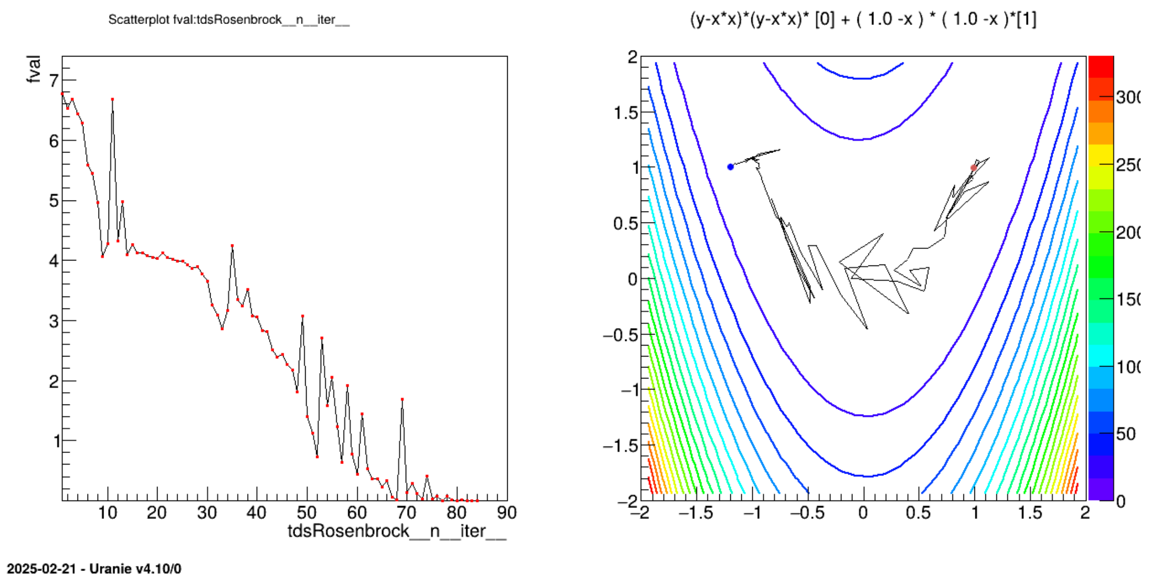


Figure XIV.79: Graph of the macro "optimizeCodeRosenbrockRowRecreate.C"

XIV.7.7.4 Console

```
Processing optimizeCodeRosenbrockRowRecreate.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 **   sLibraryName[Minuit2]
 **    sMethodName[Simplex]
 ** Problem[kMinimizeCode]
 ** input :: ivar[0/3] name[x]
 ** input :: ivar[1/3] name[y]
 ** output :: ivar[2/3] name[fval]
 ** TMultiGenCode::init _sCost[fval] _nCost[ 1]
 ** _sCost[fval]
 ********** Print state [0] option[Init]
 **   name [ x] Origin[kAttribute] Value[-1.2]
 **   name [ y] Origin[kAttribute] Value[1]
 **   name [ fval] Origin[kAttribute] Value[1.23457]
 ********** End Of Print state [0]
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${SOURCEDIR}/launCHER/souRCE/TCode.cxx] Line[1350]
 <URANIE::INFO> TCode::createDirectory Method
 <URANIE::INFO>  The directory "${RUNNINGDIR}/URANIE/UranieLauncher_1" does not exist
 <URANIE::INFO>  URANIE creates it for you
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>
 ********** Print state [84] option[Final]
 **   name [ x] Origin[kAttribute] Value[0.995289]
 **   name [ y] Origin[kAttribute] Value[0.99263]
 **   name [ fval] Origin[kAttribute] Value[6.34168e-05]
 ********** End Of Print state [84]
```

XIV.7.8 Macro "optimizeCodeRosenbrockRowRecreateOutputDataServer.C"

XIV.7.8.1 Objective

The objective of this macro is to perform optimisation of the rosenbrock function returned by the **rosenbrock** code (described in Section VII.2.1.3) with the values of the two attributes **x** and **y** read in an input file with "values in rows" format, input_rosenbrock_with_values_rows.dat:

```
-1.20 1.0
```

written on the fly. The output file, _output_rosenbrock_with_values_rows_.dat, is with "values in rows" format (and instanced as a TOutputFileDataServer) looks like:

```
#COLUMN_NAMES: x | y | fval | pA | pB

-1.200000e+000  1.000000e+000 6.776000e+000 1.000000e+001 1.000000e+000
```

where *pA* and *pB* are parameters of the rosenbrock function. *X* and *Y* are the values of attributes *x* and *y*, *fval* the cost variable.

XIV.7.8.2 Macro Uranie

```
{

  TString sIn = TString("input_rosenbrock_with_values_rows.dat");

  // Define the DataServer and add the two attributes
  TDataServer *tdsRosenbrock = new TDataServer("tdsRosenbrock", "Optimize Code externe ↩
      Rosenbrock via TDataServer");
  tdsRosenbrock->addAttribute(new TAttribute("x", -1.5, 1.5));
  tdsRosenbrock->addAttribute(new TAttribute("y", -1.5, 1.5));

  // The x attribute of the use case
  tdsRosenbrock->getAttribute("x")->setDefaultValue(-1.2);
  tdsRosenbrock->getAttribute("x")->setStepValue(0.01);
  tdsRosenbrock->getAttribute("x")->setFileKey(sIn,"x","",TAttributeFileKey::kNewRow);

  // The y attribute of the use case
  tdsRosenbrock->getAttribute("y")->setDefaultValue(1.);
  tdsRosenbrock->getAttribute("y")->setStepValue(0.01);
  tdsRosenbrock->getAttribute("y")->setFileKey(sIn,"y","",TAttributeFileKey::kNewRow);


  // The output file of the code where values are in row
  TOutputFileDataServer *fOutputFile = new TOutputFileDataServer(" ↩
      _output_rosenbrock_with_values_rows_.dat");
  fOutputFile->addAttribute(new TAttribute("fval"));

  // Create an TCode object for my code
  TCode *myRosenbrockCode = new TCode(tdsRosenbrock, "rosenbrock -r");
  // The working directory to launch the code

  // Add the output file
  myRosenbrockCode->addOutputFile( fOutputFile );

  // Graph
  TCanvas  *Canvas = new TCanvas("c1", "Graph for the Macro  ↩
      optimizeCodeRosenbrockRowRecreateOutputDataServer",5,64,1270,667);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(2,1);
  pad->cd(1);

  // Create an TOptimizer object from TDataServer and TCode objects
  TOptimizer * topt = new TOptimizer(tdsRosenbrock, myRosenbrockCode);

  topt->setMethod(TOptimizer::kSimplex);
  //topt->setTolerance(0.1);
  topt->setMaxIterations(130);

  topt->optimize("same");
  tdsRosenbrock->exportData("_tds_rosenbrock_.dat");

  pad->cd(2);
  TF2 * frosenbrok = new TF2("fcnRosenbrock","(y-x*x)*(y-x*x)* [0] + ( 1.0 -x ) * ( 1.0 -x  ↩
      )*[1]",-2.0, 2.0, -2.0,2.0);
  frosenbrok->SetParameter(0,10.0);
  frosenbrok->SetParameter(1,1.0);
  frosenbrok->Draw("cont1z");
  tdsRosenbrock->draw("y:x", "", "same1");

  tdsRosenbrock->getTuple()->SetMarkerColor(4);
  tdsRosenbrock->getTuple()->SetMarkerStyle(8);
  tdsRosenbrock->getTuple()->SetMarkerSize(.90);
  tdsRosenbrock->Draw("y:x", Form("%s==1", tdsRosenbrock->getIteratorName()), "psame");
  tdsRosenbrock->getTuple()->SetMarkerColor(50);
  tdsRosenbrock->Draw("y:x", Form("%s==%d", tdsRosenbrock->getIteratorName(), tdsRosenbrock ↩
      ->getNPatterns()), "psame");

}
```

The `TAttribute` objects **x** and **y** are linked to the input file `input_rosenbrock_with_values_rows.dat`:

```
TDataServer *tdsRosenbrock = new TDataServer("tdsRosenbrock", "Optimize Code external  ↩
    Rosenbrock via TDataServer");
tdsRosenbrock->addAttribute(new TAttribute("x", -1.5, 1.5));
tdsRosenbrock->addAttribute(new TAttribute("y", -1.5, 1.5));
```

```
tdsRosenbrock->getAttribute("x")->setDefaultValue(-1.2);
tdsRosenbrock->getAttribute("x")->setStepValue(0.01);
tdsRosenbrock->getAttribute("x")->setFileKey(sJDDReference,"x","",TAttributeFileKey:: ↩
    kNewRow);

tdsRosenbrock->getAttribute("y")->setDefaultValue(1.);
tdsRosenbrock->getAttribute("y")->setStepValue(0.01);
tdsRosenbrock->getAttribute("y")->setFileKey(sJDDReference,"y","",TAttributeFileKey:: ↩
    kNewRow);
```

Instantiating the output file:

```
TOutputFileDataServer *fOutputFile = new TOutputFileDataServer(" ↩
    _output_rosenbrock_with_values_rows_.dat");
```

The cost variable is added to the output file as a new `TAttribute`:

```
fOutputFile->addAttribute(new TAttribute("fval"));
```

Instantiating the `TCode` object, the **rosenbrock** code is launched with the *-r* option. The input file searched by the code will then be with type "values in rows":

```
TCode *myRosenbrockCode = new TCode(tdsRosenbrock, "rosenbrock -v -r");
```

The `TOptimizer` object is initialised with the `TDataServer` containing data and the `TCode` object. The optimisation is built with the *Simplex* method:

```
TOptimizer * topt = new TOptimizer(tdsRosenbrock, myRosenbrockCode);
topt->setMethod(TOptimizer::kSimplex);
topt->optimize("same");
```

The `TDataServer` is exported in an ASCII file:

```
tdsRosenbrock->exportData("_tds_rosenbrock_.dat");
```

XIV.7.8.3 Graph

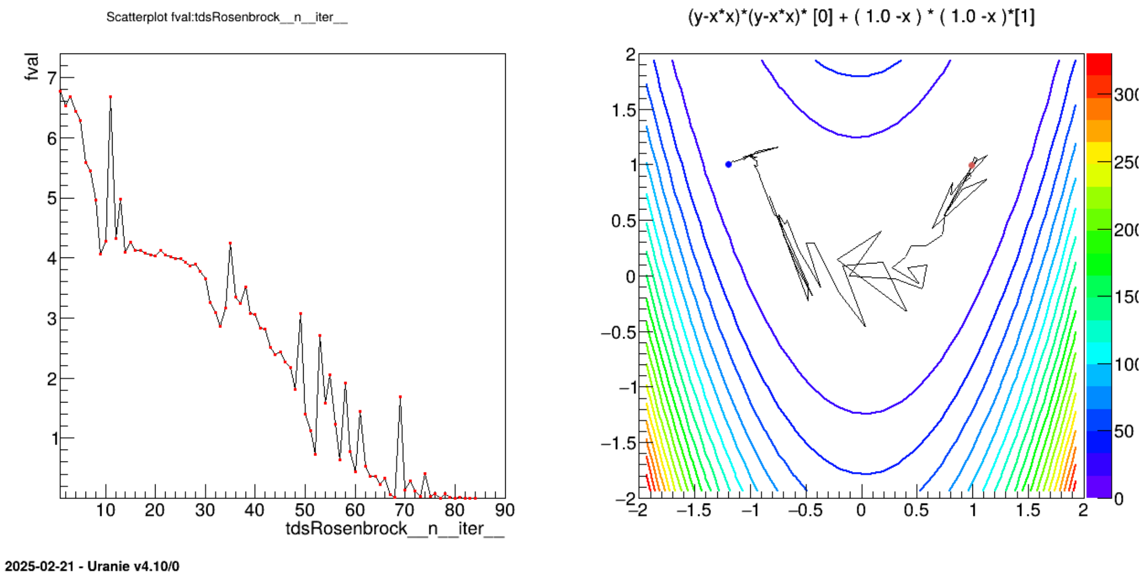


Figure XIV.80: Graph of the macro "`optimizeCodeRosenbrockRowRecreateOutputDataServer.C`"

XIV.7.8.4 Console

```
Processing optimizeCodeRosenbrockRowRecreateOutputDataServer.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 **   sLibraryName[Minuit2]
 **    sMethodName[Simplex]
 ** Problem[kMinimizeCode]
 ** input :: ivar[0/3] name[x]
 ** input :: ivar[1/3] name[y]
 ** output :: ivar[2/3] name[fval]
 ** TMultiGenCode::init _sCost[fval] _nCost[ 1]
 ** _sCost[fval]
 ********** Print state [0] option[Init]
 **   name [ x] Origin[kAttribute] Value[-1.2]
 **   name [ y] Origin[kAttribute] Value[1]
 **   name [ fval] Origin[kAttribute] Value[1.23457]
 ********** End Of Print state [0]
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${SOURCEDIR}/launCHER/souRCE/TCode.cxx] Line[1350]
 <URANIE::INFO> TCode::createDirectory Method
 <URANIE::INFO>  The directory "${RUNNINGDIR}/URANIE/UranieLauncher_1" does not exist
 <URANIE::INFO>  URANIE creates it for you
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>
 ********** Print state [84] option[Final]
 **   name [ x] Origin[kAttribute] Value[0.995289]
 **   name [ y] Origin[kAttribute] Value[0.99263]
 **   name [ fval] Origin[kAttribute] Value[6.34168e-05]
 ********** End Of Print state [84]
```

XIV.7.9 Example of optimisation with a code that can compute several values at each run

XIV.7.9.1 Objective

Warning As Uranie does not provide any external code that is able to compute several output values for several input sets, the following code is just given as an example to illustrate the behaviour of the `TOptimizer` class.

This part introduces the logic of the upcoming examples in Section XIV.7.10 and Section XIV.7.11. We consider here that the **myCode** external code depends on three parameters, a, b and c, and we want to find the values of these parameters that best fit a reference set of values.

XIV.7.9.2 Macro Uranie

```
{
  TDataServer * tds = new TDataServer();                                      ❶[1]
  tds->addAttribute( new TAttribute("a", -5.0, 5.0));
  tds->addAttribute( new TAttribute("b", -1.0, 1.0));
  tds->addAttribute( new TAttribute("c", -5.0, 5.0));

  // The reference input file                                  ❷[2]
  TString sJDDReference = gSystem->pwd();
  sJDDReference += TString("/myCode_param.in");
  tds->getAttribute("a")->setFileKey(sJDDReference,"a");
  tds->getAttribute("b")->setFileKey(sJDDReference,"b");
  tds->getAttribute("c")->setFileKey(sJDDReference,"c");

  tds->getAttribute("a")->setDefaultValue(0.200);
  tds->getAttribute("a")->setStepValue(0.01);
  tds->getAttribute("b")->setDefaultValue(0.200);
  tds->getAttribute("b")->setStepValue(0.01);
  tds->getAttribute("c")->setDefaultValue(0.200);
  tds->getAttribute("c")->setStepValue(0.01);

  // The output file of the code                                ❸[3]
  TOutputFileRow *fout = new TOutputFileRow("myCode_output.dat");
  // The attribute in the output file
  fout->addAttribute(new TAttribute("yhat"));

  TCode *mycode = new TCode (tds, "myCode >> /dev/null");       ❹[4]
  mycode->addOutputFile( fout );

  // Read the reference dataserver, that contains the "ystar" attribute
  TDataServer *tdsref = new TDataServer("TDSRef", "Objectives values"); ❺[5]
  tdsref->fileDataRead("myCode_ref.dat");

  // Definition of the optimizer
  TOptimizer * topt = new TOptimizer(tds, mycode);              ❻[6]
  topt->addObjective( "obj1", tdsref, "ystar", fout, "yhat");   ❼[7]
  topt->optimize();                                             ❽[8]
}
```

Optimisation of a code that can compute several values at each run

❶ Definition of the dataserver and creation of the parameters to find as new attributes;

❷ Definition of the input file of the code (myCode_param.in), and definition for each parameter of its name in the input file, default value and step value;

❸ Definition of the output file of the code where to find the computed output "yhat", which is a vector of values;

❹ Definition of the code;

❺ Creation of the dataserver that contains the reference values, and read the values from file "`myCode_ref.dat`";

❻ Definition of the `TOptimizer` object from the dataserver and the code;

❼ Addition of an objective named "obj1", where the values of "yhat" retrieved from output file "`myCode_output.dat`" (`fout`) will be compared to the values of "ystar" contained in the dataserver `tdsref`;

❽ Run of the optimisation process.

XIV.7.10 Macro "optimizeRosenbrockMulti.C"

XIV.7.10.1 Objective

Considering the `rosenbrock` code example. We have generated a set of values with parameters a and b fixed, and stored both input and output values in the file `rosenbrock_ref.dat`:

```
#NAME: testRosenbrock
#TITLE: Rosenbrock
#DATE: mar sep 25 15:30:20 CEST 2012
#COLUMN_NAMES: x1| x2| y

-1.350076223e+00 3.365490640e-01 1.767170212e+01
1.183157479e+00 -6.425784534e-01 1.258177769e+01
-9.634946792e-01 -1.301159335e+00 2.262238374e+01
1.522053698e+00 -4.413296049e-01 2.336439261e+01
-5.921963897e-01 -1.189104788e+00 1.218314329e+01
7.477007261e-01 4.940760535e-01 1.399771741e-01
1.461164459e+00 9.121608682e-01 4.911363513e+00
3.793243287e-01 1.623933752e+00 7.342092216e+00
-1.713026222e+00 -4.397660890e-01 4.887720411e+01
1.228036214e+00 1.425142762e-01 5.698252451e+00
-1.327069995e+00 1.133228437e+00 1.201323326e+01
1.465765779e+00 -4.783690687e-01 2.113471528e+01
-2.659000948e-01 -3.717979765e-01 3.792427072e+00
1.326644911e+00 -6.113681370e-02 1.016286658e+01
-5.986060012e-01 -1.611296639e+00 1.674935948e+01
-8.026301529e-01 3.762660855e-02 7.602799958e+00
```

XIV.7.10.2 Creation of the function to minimise

The `rosenbrock` code is only able to take one set of parameters (a,b) at each call, and returns only one value each time. In order for Uranie to find the parameters that best fit the results above, it is necessary to add a level of "code" that will be able to compute a set of outputs from a set of inputs. This new "code" will be the Uranie macro below:

```
{
  // Definition of the reference dataserver
  TDataServer *tdsMulti = new TDataServer("tdsMulti", "Objectives values");  ↩
                    ❶[1]
  tdsMulti->fileDataRead("rosenbrock_ref.dat");
  TString sinputfile ="input_rosenbrock_with_keys.dat";
  tdsMulti->getAttribute("x1")->setFileKey(sinputfile, "x");
  tdsMulti->getAttribute("x2")->setFileKey(sinputfile, "y");

  // Definition of the output file of the code  ↩
                                                  ❷[2]
  TOutputFileRow *fout = new TOutputFileRow("_output_rosenbrock_with_values_rows_.dat");
  fout->addAttribute(new TAttribute("yhat"), 3);

  // Definition of the code  ↩
                                                                   ❸[3]
  TCode *mycode = new TCode(tdsMulti, "rosenbrock -k >> /dev/null");
  mycode->addOutputFile(fout);

  // Launcher on the TDS Ref  ↩
                                                                  ❹[4]
  TLauncher * tl = new TLauncher(tdsMulti, mycode);
  tl->setDrawProgressBar(kFALSE);
  tl->run();

  tdsMulti->exportData("_rosenbrock_multi_.dat","x1:x2:yhat");  ↩
                                           ❺[5]
}
```

Creation of a macro able to compute a set of outputs from a set of inputs

❶ Creation of the dataserver that will store the generated values, and addition of the attributes x1 and x2 that will be read in the input file "input_rosenbrock_with_keys.dat";

❷ Definition of the output file of the code, "_output_rosenbrock_with_values_rows_.dat", and definition of the output attribute, that will be called "yhat", and that is present at the 3rd column of the file;

❸ Definition of the code to run and its output file;

❹ Creation of a launcher to run the code, and execution of the computation;

❺ Export the results of the macro, *i.e.* the set of outputs generated for the Rosenbrock code. The selected output variables are x1, x2 and yhat.

XIV.7.10.3 Macro Uranie

Now that we have an external code (the ROOT macro just defined above), we can use the same optimisation scheme than the one described in Section XIV.7.9, but the code to run will be the ROOT macro execution:

```
{

  // Definition of the dataserver
  TDataServer * tds = new TDataServer();

  // Definition of the attributes
  TString sJDDReference = "input_rosenbrock_with_keys.dat";
  tds->addAttribute( new TAttribute("a", 0.0, 5.0));
  tds->getAttribute("a")->setFileKey(sJDDReference,"a");
  tds->getAttribute("a")->setDefaultValue(1.0);
  tds->getAttribute("a")->setStepValue(0.1);
  tds->addAttribute( new TAttribute("b", 0.0, 5.0));
  tds->getAttribute("b")->setFileKey(sJDDReference,"b");
  tds->getAttribute("b")->setDefaultValue(2.0);
  tds->getAttribute("b")->setStepValue(0.1);

  // Definition of the output file of the code
  TOutputFileRow *fout = new TOutputFileRow("_rosenbrock_multi_.dat");
  fout->addAttribute(new TAttribute("yhat"), 3);

  // OS abstraction
  string to_null =
    string(gSystem->GetBuildArch()) == "win64" ? " > NUL" : " >> /dev/null";

  // Definition of the code
  TCode *mycode = new TCode(tds, "root -b -l -q rosenbrock_multi.C" + to_null);
  mycode->addOutputFile(fout);
  mycode->addInputFile("rootlogon.C");
  mycode->addInputFile("rosenbrock_ref.dat");
  mycode->addInputFile("rosenbrock_multi.C");

  // Definition of the reference dataserver
  TDataServer *tdsref = new TDataServer("tdsref", "Objectives values");
  tdsref->fileDataRead("rosenbrock_ref.dat");

  // Definition of the optimizer
  TOptimizer *topt = new TOptimizer(tds, mycode);
  topt->addObjective("obj1", tdsref, "y", fout, "yhat");
  topt->optimize();

  TCanvas *c = new TCanvas();
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  tds->draw("b:a","","lp");

}
```

Creation of the dataserver that will store the searched parameters values, and addition of the attributes a and b that will be read in the input file "input_rosenbrock_with_keys.dat". Note that these definitions introduce the domains of research of a and b as the interval [0 ; 5], the start values as 1. for a and 2. for b, and the step value as 0.1.

```
// Definition of the dataserver
TDataServer * tds = new TDataServer();
```

Definition of the output file of the code, "_rosenbrock_multi_.dat" (generated by the macro rosenbrock_multi.C), and definition of the output attribute, *"yhat"*:

```
// Definition of the output file of the code
TOutputFileRow *fout = new TOutputFileRow("_rosenbrock_multi_.dat");
fout->addAttribute(new TAttribute("yhat"), 3);
```

Definition of the code to run (the root command), its output file, and the files it needs to be able and run the computation. Note that ROOT is run with options *"-l"* not to display splash screen, and *"-q"* to exit after execution (otherwise the macro execution would fail):

```
// Definition of the code
TCode *mycode = new TCode(tds, "root -l -q rosenbrock_multi.C >> /dev/null");
mycode->addOutputFile(fout);
mycode->addInputFile("rootlogon.C");
mycode->addInputFile("rosenbrock_ref.dat");
mycode->addInputFile("rosenbrock_multi.C");
```

Creation of the dataserver tdsref for the reference values, and importation of these values:

```
// Definition of the reference dataserver
TDataServer *tdsref = new TDataServer("tdsref", "Objectives values");
tdsref->fileDataRead("rosenbrock_ref.dat");
```

Definition of the optimizer object, addition of an objective from the *"y"* attribute of tdsref and the *"yhat"* attribute read from the file fout, and run of the optimisation process:

```
// Definition of the optimizer
TOptimizer *topt = new TOptimizer(tds, mycode);
topt->addObjective("obj1", tdsref, "y", fout, "yhat");
topt->optimize();
```

Plot of the evolution of searched parameters a and b throw iterations:

```
TCanvas *c = new TCanvas();
tds->draw("b:a","","lp");
```

XIV.7.10.4 Graph

We can see in the graphic below that the parameters a and b converge from the point (1.;2.) to the point (3.;2.), where the values used to generate the data stored in the `rosenbrock_ref.dat` file.

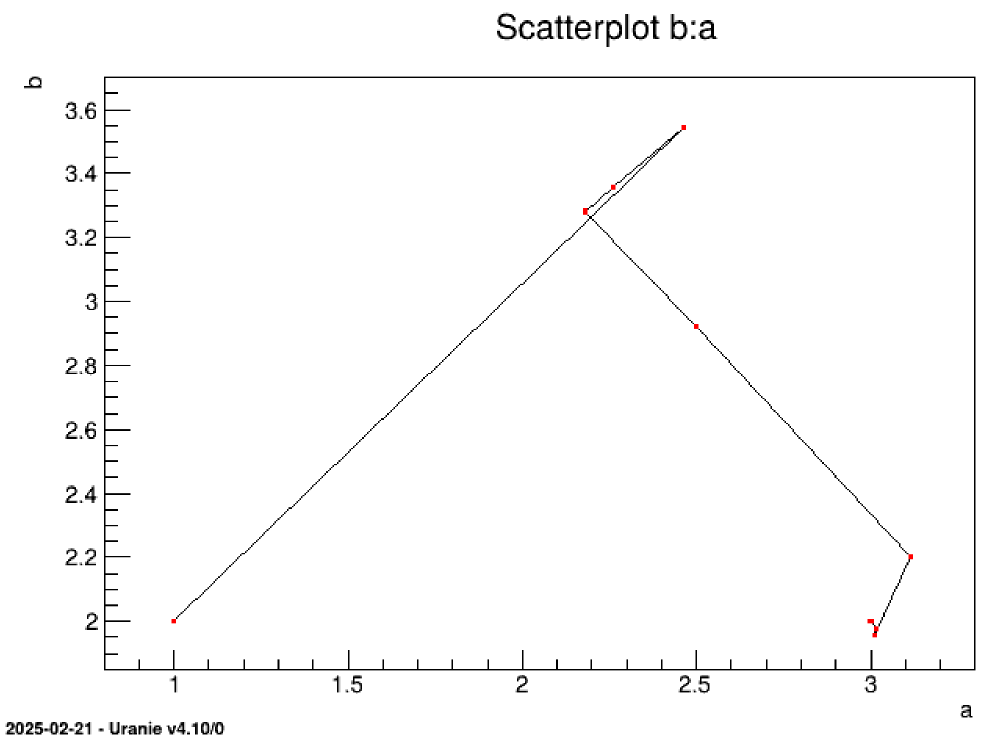


Figure XIV.81: Evolution of searched parameters a and b throw iterations

XIV.7.10.5 Console

```
Processing optimizeRosenbrockMulti.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025



 ** addObjective ystar(y) in TDS (tdsref) yhat(yhat) in the output file(_rosenbrock_multi_. ↩
    dat)
 **   sLibraryName[Minuit2]
 **    sMethodName[Migrad]
 ** Problem[kSumOfSquare]
 ** _ninput [ 2]
 ** _noutput [ 1]
 ** Attribute[yhat]-- Del
 ** Attribute[b]
 ** Attribute[a]
 ** indx[0] Name[obj1]
 ** list of Objectives size(1) _dSumOfWeight(0)
 ** Objective i[0/1] Name[obj1] weight( 1)
 ** End Of list of Objectives _dSumOfWeight[1]
 ** _sCriteria [ obj1]

*************************************
 *** TMultiGenSumOfSquares::init
 *** TMultiGenSumOfSquares::clean
 *** End Of TMultiGenSumOfSquares::init
*************************************
 ** _sCost[obj1]
 ********** Print state [0] option[Init]
 **   name [ a] Origin[kAttribute] Value[1]
 **   name [ b] Origin[kAttribute] Value[2]
 **   name [ obj1] Origin[kAttribute] Value[1.23457]
 ********** End Of Print state [0]
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${SOURCEDIR}/launCHER/souRCE/TCode.cxx] Line[1350]
 <URANIE::INFO> TCode::createDirectory Method
 <URANIE::INFO>  The directory "${RUNNINGDIR}/URANIE/UranieLauncher_1" does not exist
 <URANIE::INFO>  URANIE creates it for you
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>
 ********** Print state [53] option[Final]
 **   name [ a] Origin[kAttribute] Value[2.99971]
 **   name [ b] Origin[kAttribute] Value[1.99974]
 **   name [ obj1] Origin[kAttribute] Value[3.69704e-06]
 ********** End Of Print state [53]
```

XIV.7.11 Macro "optimizeRosenbrockError.C"

XIV.7.11.1 Objective

If we consider now that the external code (the Root macro in our case) cannot return the computed values anymore, but only an error (the quadratic error in our case) between the reference values and the estimated values on the reference TDataServer, we are back in the function optimisation situation, where the function to minimise is the error function, and its input parameters the parameters of the function.

XIV.7.11.2 Creation of the function to minimise

To get such a case, we just modify the rosenbrock_multi.C macro to become the rosenbrock_error.C code below:

```
{
  // Definition of the reference dataserver
  TDataServer *tdsError = new TDataServer("tdsError", "Objectives values");          ❶[1]
  tdsError->fileDataRead("rosenbrock_ref.dat");

  TString sinputfile ="input_rosenbrock_with_keys.dat";
  tdsError->getAttribute("x1")->setFileKey(sinputfile, "x");
  tdsError->getAttribute("x2")->setFileKey(sinputfile, "y");


  // Definition of the output file of the code                                        ❷[2]
  TOutputFileRow *fout = new TOutputFileRow("_output_rosenbrock_with_values_rows_.dat");
  fout->addAttribute(new TAttribute("yhat"), 3);
  // Definition of the code                                                          ❸[3]
  TCode *mycode = new TCode(tdsError, "rosenbrock -k >> /dev/null");
  mycode->addOutputFile(fout);


  // Launcher on the TDS Ref                                                         ❹[4]
  TLauncher * tl = new TLauncher(tdsError, mycode);
  tl->setDrawProgressBar(kFALSE);
  tl->run();

  // Export the reference values for the (a,b) parameters                           ❺[5]
  tdsError->exportData("_output_rosenbrock_references_.dat","x1:x2:yhat:y");

  // Compute the L1, L2 and LMax norms for the (a,b) parameters
  tdsError->addAttribute("squareDiff", "(yhat-y)^2.");                               ❻[6]
  tdsError->computeStatistic("squareDiff");

  Double_t normL2 = tdsError->getAttribute("squareDiff")->getMean();                 ❼[7]

  ofstream outFile("_rosenbrock_error_.dat",ios::out);                               ❽[8]
  outFile << "normL2 = " << normL2 << ";" << endl;
  outFile << endl;
  outFile.close();
}
```

Creation of a macro able to compute a set of outputs from a set of inputs

❶ Creation of the dataserver that will store the generated values, and addition of the attributes *x1* and *x2* that will be read in the input file "`input_rosenbrock_with_keys.dat`";

❷ Definition of the output file of the code, "`_output_rosenbrock_with_values_rows_.dat`", and definition of the output attribute, that will be called *"yhat"*, and that is present at the 3rd column of the file;

❸ Definition of the code to run and its output file;

❹ Creation of a launcher to run the code, and execution of the computation;

❺ Export the results of the macro, *i.e.* the set of outputs generated for the Rosenbrock code, in the file "`_output_rosenbrock_references_.dat`". The selected output variables are x1, x2, y and yhat.

❻ Compute a new attribute that will be used to generate the norms of the error, `squareDiff` which is the vector of square differences between reference and computed vectors. Then compute the statistics for this attribute.

❼ Create the scalar value of the L2 norms of the error, thanks to the attributes previously defined.

❽ Export the results of the macro, *i.e.* the set of outputs generated for the `rosenbrock` code. The selected output variables are *x1*, *x2* and *yhat*.

XIV.7.11.3 Macro Uranie

We can now use this macro as an external code we want to minimise the output:

```
{

  // Definition of the dataserver
  TDataServer * tds = new TDataServer();

  // Definition of the attributes
  TString sJDDReference = "input_rosenbrock_with_keys.dat";
  tds->addAttribute( new TAttribute("a", 0.0, 5.0));
  tds->getAttribute("a")->setFileKey(sJDDReference,"a");
  tds->getAttribute("a")->setDefaultValue(1.0);
  tds->getAttribute("a")->setStepValue(0.1);
  tds->addAttribute( new TAttribute("b", 0.0, 5.0));
  tds->getAttribute("b")->setFileKey(sJDDReference,"b");
  tds->getAttribute("b")->setDefaultValue(2.0);
  tds->getAttribute("b")->setStepValue(0.1);

  // Definition of the output file of the code
  TOutputFileKey *fout = new TOutputFileKey("_rosenbrock_error_.dat");
  fout->addAttribute(new TAttribute("normL2"));

  // OS abstraction
  string to_null =
    string(gSystem->GetBuildArch()) == "win64" ? " > NUL" : " >> /dev/null";

  // Definition of the code
  TCode *mycode = new TCode(tds, "root -b -l -q rosenbrock_error.C" + to_null);
  mycode->addOutputFile(fout);
  mycode->addInputFile("rootlogon.C");
  mycode->addInputFile("rosenbrock_ref.dat");
  mycode->addInputFile("rosenbrock_error.C");

  // Definition of the optimizer
  TOptimizer *topt = new TOptimizer(tds, mycode);
  topt->setTolerance(1e-5);
  topt->optimize();

  TCanvas *c = new TCanvas();
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  tds->draw("b:a","","lp");

}
```

Creation of the dataserver that will store the searched parameters values, and addition of the attributes a and b that will be read in the input file "input_rosenbrock_with_keys.dat".

```
// Definition of the dataserver
TDataServer * tds = new TDataServer();

// Definition of the attributes
TString sJDDReference = "input_rosenbrock_with_keys.dat"
```

Definition of the output file of the code, "_rosenbrock_error_.dat" (generated by the macro rosenbrock_error.C), and definition of the output attribute that will be got them from this file, *"normL2"*;

```
// Definition of the output file of the code
TOutputFileKey *fout = new TOutputFileKey("_rosenbrock_error_.dat");
fout->addAttribute(new TAttribute("normL2"));
```

Definition of the code to run (the root command), its output file, and the files it needs to be able to run the computation. Note that ROOT is run with options *"-l"* not to display splash screen, and *"-q"* to exit after execution (otherwise the macro execution would fail).

```
// Definition of the code
TCode *mycode = new TCode(tds, "root -l -q rosenbrock_error.C >> /dev/null");
```

Definition of the optimizer object, setting of the tolerance to 1e-5, and run of the optimisation process:

```
// Definition of the optimizer
TOptimizer *topt = new TOptimizer(tds, mycode);
topt->setTolerance(1e-5);
topt->optimize();
```

The remaining lines are dealing with the plotting of the evolution of searched parameters a and b throw iterations.

XIV.7.11.4 Graph

We can see in the graphic below that the parameters a and b converge from the point (1.;2.) to the point (3.;2.), where the values used to generate the data stored in the `rosenbrock_ref.dat` file.

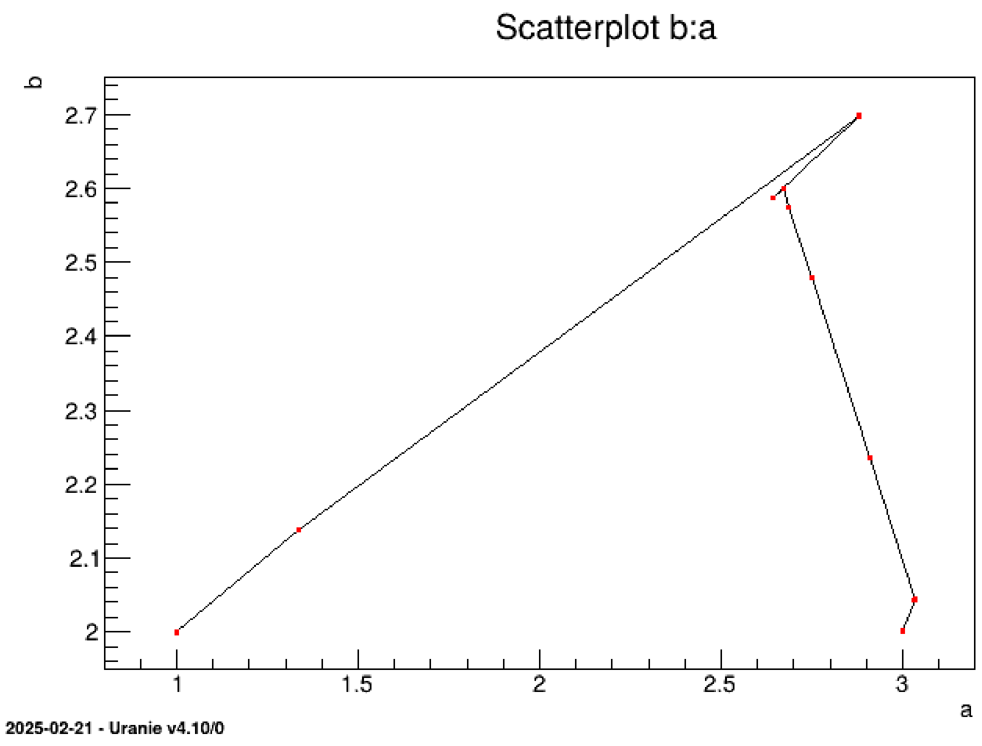


Figure XIV.82: Evolution of searched parameters a and b throw iterations

XIV.7.11.5 Console

```
Processing optimizeRosenbrockError.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025
```

```
**   sLibraryName[Minuit2]
**    sMethodName[Migrad]
** Problem[kMinimizeCode]
** input :: ivar[0/3] name[a]
** input :: ivar[1/3] name[b]
** output :: ivar[2/3] name[normL2]
** TMultiGenCode::init _sCost[normL2] _nCost[ 1]
** _sCost[normL2]
********** Print state [0] option[Init]
**   name [ a] Origin[kAttribute] Value[1]
**   name [ b] Origin[kAttribute] Value[2]
**   name [ normL2] Origin[kAttribute] Value[1.23457]
********** End Of Print state [0]
<URANIE::INFO>
<URANIE::INFO> *** URANIE INFORMATION ***
<URANIE::INFO> *** File[${SOURCEDIR}/launCHER/souRCE/TCode.cxx] Line[1350]
<URANIE::INFO> TCode::createDirectory Method
<URANIE::INFO>  The directory "${RUNNINGDIR}/URANIE/UranieLauncher_1" does not exist
<URANIE::INFO>  URANIE creates it for you
<URANIE::INFO> *** END of URANIE INFORMATION ***
<URANIE::INFO>
********** Print state [68] option[Final]
**   name [ a] Origin[kAttribute] Value[3]
**   name [ b] Origin[kAttribute] Value[2.00001]
**   name [ normL2] Origin[kAttribute] Value[2.75242e-10]
********** End Of Print state [68]
```

XIV.8 Macros Relauncher

The idea of this section is to show the basic usage of many of the classes defined in Chapter VIII, applied either on the `flowrate` functions or the **flowrate** code, whose purpose and behaviour have been already introduced in Section IV.1.2.1. All the following examples will load a tiny set of points which is gathered in the file called `flowrateUniformDesign.dat` (already introduced in Section XIV.2.4.1).

XIV.8.1 Macro "`relauncherFunctionFlowrateCInt.C`"

XIV.8.1.1 Objective

The goal of this macro is to show how to handle (in the most simple way) a C++-written function, compliant with the ROOT (CINT) format. This function has been presented, at least its equation (see Equation IV.1) and would be interfaced through the `TCIntEval` class in the Relauncher module (which means that we'll use the function database from ROOT's catalog, see Section I.2.5 for more explanations). As this class is usually considered not thread-safe, it can only be used with a `TSequentialRun` runner.

XIV.8.1.2 Macro

```
void flowrateModel(double *x, double *y)
{
  double drw = x[0], dr  = x[1];
  double dtu = x[2], dtl = x[3];
  double dhu = x[4], dhl = x[5];
  double dl  = x[6], dkw = x[7];
  double dnum = 2.0 * TMath::Pi() * dtu * ( dhu -dhl);
  double dlnronrw = TMath::Log( dr / drw);
  double dden = dlnronrw * ( 1.0 +  ( 2.0 * dl * dtu ) / ( dlnronrw * drw * drw * dkw) + ↩
      dtu / dtl );

  y[0] = dnum / dden;
}

void relauncherFunctionFlowrateCInt()
{

  // Create the TDataServer
  TDataServer *tds = new TDataServer("foo","test");
  tds->fileDataRead("flowrateUniformDesign.dat");

  // Get the attributes
  TAttribute *rw = tds->getAttribute("rw");
  TAttribute *r = tds->getAttribute("r");
  TAttribute *tu = tds->getAttribute("tu");
  TAttribute *tl = tds->getAttribute("tl");
  TAttribute *hu = tds->getAttribute("hu");
  TAttribute *hl = tds->getAttribute("hl");
  TAttribute *l = tds->getAttribute("l");
  TAttribute *kw = tds->getAttribute("kw");

  // Create the output attribute
  TAttribute *yhat = new TAttribute("yhat");

  // Constructing the code
  TCIntEval mycode("flowrateModel");
  mycode.setInputs(8, rw, r, tu, tl, hu, hl, l, kw); // Adding the input attributes
  mycode.addOutput(yhat); // Adding the output attributes

  // Create the sequential runner
  TSequentialRun run(&mycode);
  run.startSlave(); //Start the master (necessary even for a sequential)
  if (run.onMaster())
  {
      TLauncher2 lanceur(tds, &run);

      // resolution
      lanceur.solverLoop();
      run.stopSlave(); // Stop the slaves (necessary even for a sequential)
  }

  // Draw the result
  TCanvas *can = new TCanvas("pouet","foo",1);
  tds->Draw("yhat:rw","","colZ");


}
```

The first part of the macro is the definition of the `flowrateModel` function, already discussed throughout this documentation. The dataserver object is then created and filled using the database file and pointers to the corresponding input attributes are created, along with the new attribute for the output provided by the function. The following part is then specific to the Relauncher organisation: a `TCIntEval` object is created with the function as only argument. Both the input and output attributes are provided (here in a contracted way for input, but it could have been done one-by-one, as for output).

```
// Constructing the code
```

```
TCIntEval mycode(flowrateModel);
mycode.setInputs(8, rw, r, tu, tl, hu, hl, l, kw); // Adding the input attributes
mycode.addOutput(yhat); // Adding the output attributes
```

The following part is the heart of the relauncher strategy: the assessor is provided to the chosen runner, which should always start the slaves (even in the case of a sequential one like here). On the main CPU, the master is created as well (with the dataserver and the runner) and the resolution is requested.

```
// Create the sequential runner
TSequentialRun run(&mycode);
run.startSlave(); //Start the master (necessary even for a sequential)
if (run.onMaster())
{
  TLauncher2 lanceur(tds, &run);

  // resolution
  lanceur.solverLoop();
  run.stopSlave(); // Stop the slaves (necessary even for a sequential)
}
```

Once this is done, the slaves are stopped and the results is displayed for cross-check in the following subsection.

XIV.8.1.3 Graph

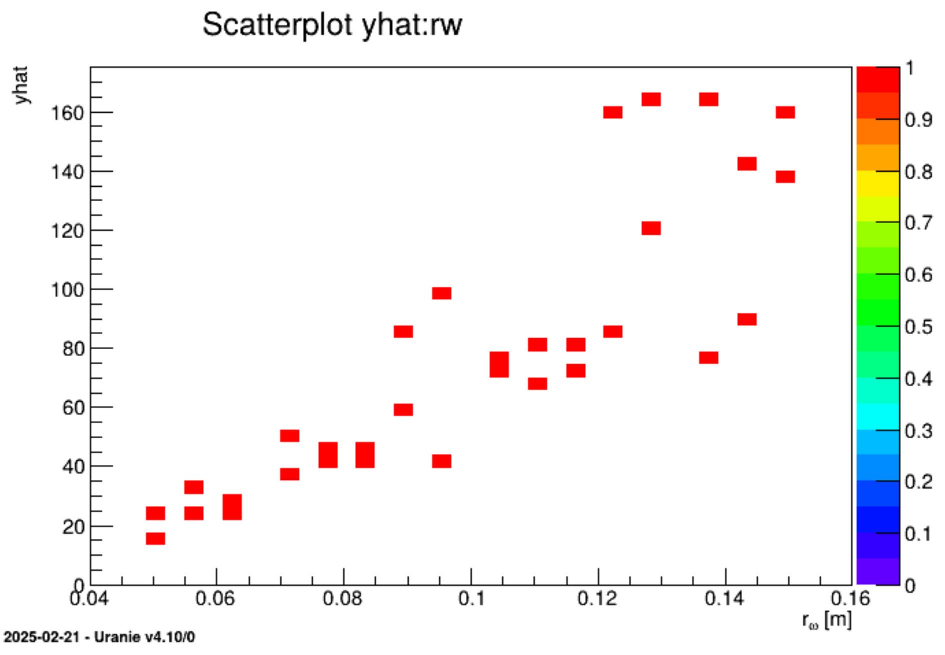


Figure XIV.83: Representation of the output as a function of the first input with a colZ option

XIV.8.2 Macro "relauncherFunctionFlowrateCJit.C"

XIV.8.2.1 Objective

The goal of this macro is to show how to handle the C++-written function using a pointer to the function (and not the name as for the macro in Section XIV.8.1). This function has been presented, at least its equation (see Equation IV.1) and would be interface through the `TCJitEval` class in the Relauncher module. A special discussion will be held in the next few lines about the way the compilation is done.

XIV.8.2.2 Macro

```
#include "TCanvas.h"
#include "TMath.h"
#include "TSystem.h"
#include "TDataServer.h"
#include "TSequentialRun.h"
#include "TAttribute.h"
#include "TLauncher2.h"
#include "TCJitEval.h"

using namespace URANIE::DataServer;
using namespace URANIE::Relauncher;

void flowrateModel(double *x, double *y)
{
  double drw = x[0], dr  = x[1];
  double dtu = x[2], dtl = x[3];
  double dhu = x[4], dhl = x[5];
  double dl  = x[6], dkw = x[7];

  double dnum = 2.0 * TMath::Pi() * dtu * ( dhu -dhl);
  double dlnronrw = TMath::Log( dr / drw);
  double dden = dlnronrw * ( 1.0 +  ( 2.0 * dl * dtu ) / ( dlnronrw * drw * drw * dkw) +  ↩
      dtu / dtl );

  y[0] = dnum / dden;
}

// void relauncherFunctionFlowrateCJit() // For ACLIC (meaning CINT) compilation
int main() // For standalone compilation
{

  // Create the TDataServer
  TDataServer *tds = new TDataServer("foo","test");
  tds->fileDataRead("flowrateUniformDesign.dat");

  // Get the attributes
  TAttribute *rw = tds->getAttribute("rw");
  TAttribute *r = tds->getAttribute("r");
  TAttribute *tu = tds->getAttribute("tu");
  TAttribute *tl = tds->getAttribute("tl");
  TAttribute *hu = tds->getAttribute("hu");
  TAttribute *hl = tds->getAttribute("hl");
  TAttribute *l = tds->getAttribute("l");
  TAttribute *kw = tds->getAttribute("kw");

  // Create the output attribute
  TAttribute *yhat = new TAttribute("yhat");

  // Constructing the code
  TCJitEval mycode(flowrateModel);
  mycode.setInputs(8, rw, r, tu, tl, hu, hl, l, kw); // Adding the input attribute
  mycode.addOutput(yhat); // Adding the output attribute

  // Create the sequential runner
  TSequentialRun run(&mycode);
  run.startSlave(); //Start the master (necessary even for a sequential)
  if (run.onMaster())
  {
    TLauncher2 lanceur(tds, &run);

    // resolution
    lanceur.solverLoop();
    run.stopSlave(); // Stop the slaves (necessary even for a sequential)
  }

  // Export the data
  tds->exportData("_outputFile_functionflowrate_cjit_.dat");

}
```

The very first part of the macro is slightly different from Section XIV.8.1.2 as it is now compulsory to write explicitly the include line for the pre-processors. The use of namespaces is declared along, but this time only because this is convenient not to recall these long names every time.

```
#include "TCanvas.h"
#include "TMath.h"
#include "TSystem.h"
#include "TDataServer.h"
#include "TSequentialRun.h"
#include "TAttribute.h"
#include "TLauncher2.h"
#include "TCJitEval.h"

using namespace URANIE::DataServer;
using namespace URANIE::Relauncher;
```

The definition of the `flowrateModel` function is then made right before entering the main function. At this level a break is needed to explain the risen antagonism between the two possible compilation and the way to call them. This issue is entirely described by the following two lines:

```
// void relauncherFunctionFlowrateCJit() // For ACLIC (meaning CINT) compilation
int main() // For standalone compilation
```

Let's talk about the two cases:

- ACLIC: one should remove the second line and put the first one instead (after commenting it out of course). The compilation is then done by calling:

  ```
  root -l relauncherFunctionFlowrateCJit.C+
  ```

  – PROS: this will give you the hand at the end of execution, leaving the display opened to handle plots for instance.
  – CONS: this will create several useless files and will need non-trivial manipulation if extra headers and libraries are needed.

- Standalone: leaving the macro as it is, the compilation on Linux is done by writing a line such as:

  ```
  g++ -o CJitTest relauncherFunctionFlowrateCJit.C `root-config --cflags --libs` - ↩
      L$URANIESYS/lib -lUranieDataServer  -lUranieRelauncher -I$URANIESYS/include/
  ```

  An equivalent command on Windows would be:

  ```
  cl /Fe%cd%\CJitTest /Tp relauncherFunctionFlowrateCJit.C /I%ROOTSYS%\include %ROOTSYS%\lib ↩
      \lib*.lib %URANIESYS%\lib\libUranieDataServer.lib %URANIESYS%\lib\libUranieRelauncher. ↩
      lib
  ```

- PROS: pure C++ compilation resulting in a single executable file (here *CJitTest*). It's easy to include more headers and libraries if your code needs them.
- CONS: this will not leave the display opened, unless you do so through a `TApplication` (ROOT-class).

Whatever the chosen solution, the only difference with previous macro is the assessor creation:

```
TCJitEval mycode(flowrateModel);
```

The rest it completely transparent and leads to the creation of the following plot.

XIV.8.2.3 Graph

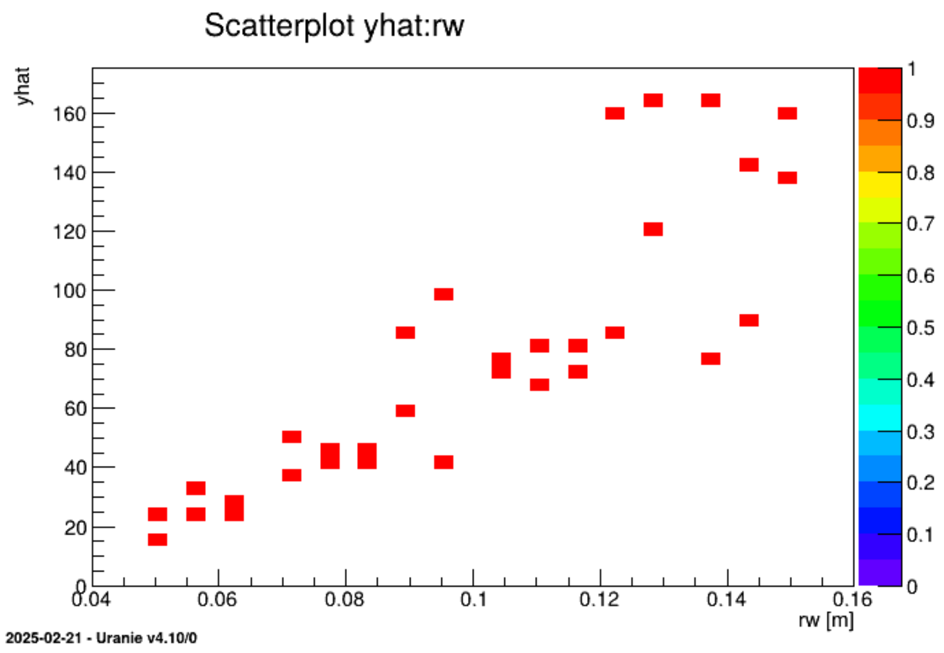


Figure XIV.84: Representation of the output as a function of the first input with a colZ option

XIV.8.3 Macro "`relauncherCJitFunctionThreadTest.C`"

XIV.8.3.1 Objective

The goal of this macro is to show how to handle thread-safe compiled function (or code) that would contain `TDataServer` objects (this is not particularly recommended, but has been requested to us). This example is only written in C++ as the CJit interface only works for this, but the idea is the same if one has a code and use the python interface. This will be further discussed below. The function is pointless, a pure illustrative toy and the results is not important as long as one sees that in one case, the macro runs smoothly while on the other hand, it crashes.

XIV.8.3.2 Macro

```
void multiply(double *x, double *y)
{
    // New dataserver reading all points in flowrate
    URANIE::DataServer::TDataServer test("test","notindir");
    test.keepFinalTuple(false); // Remove the tuple from ROOT internal list
    test.fileDataRead("flowrateUniformDesign.dat",false);

    // Dummy functions with a loop to slow down the function
    double max=-1000000;
    for(int i=0; i<test.getNPatterns(); i++)
    {
        double val = test.getValue("ystar",i);
        max = ((val>=max) ? val : max);
    }
    y[0] = max * x[0];

}

void relauncherCJitFunctionThreadTest()
{

    /* This macro can be used in two modes:
    */
    bool threaded=true;

    // If thread-safe, call this method that will take out the dataserver from ROOT ↩
        internal list
    if(threaded)
        ROOT::EnableThreadSafety();  // part of the solution

    // input and output attributes
    URANIE::DataServer::TUniformDistribution x("multiplier",1,10);
    URANIE::DataServer::TAttribute    MultMean("MultMean");

    // Interface to the compiled function above
    URANIE::Relauncher::TCJitEval eval(multiply);
    eval.addInput(&x);
    eval.addOutput(&MultMean);

    // Threaded runner
    URANIE::Relauncher::TThreadedRun run(&eval,4);
    run.startSlave();

    if( run.onMaster() )
    {
        // Global dataserver
        URANIE::DataServer::TDataServer tds("pouet","pouet_notindir");
        tds.addAttribute(&x);

        // Doe for the multiplier
        URANIE::Sampler::TSampling sam(&tds,"lhs",24);
        sam.generateSample();

        // Run the code
        URANIE::Relauncher::TLauncher2 launch(&tds, &run);
        launch.solverLoop();
        tds.getTuple()->SetScanField(-1);
        tds.scan();

        run.stopSlave();

    }
}
```

The very first part of the macro is a function that would be applied to all points of our design-of-experiments. The general context is simple: let's assume one wants to find the maximum value of a variable in a given dataset, and let's assume that this maximum should have to be scaled by some factor. We create the `multiply` function to do so, as done here:

```
void multiply(double *x, double *y)
{
    // New dataserver reading all points in flowrate
    URANIE::DataServer::TDataServer test("test","notindir");
    test.keepFinalTuple(false); // Remove the tuple from ROOT internal list
    test.fileDataRead("flowrateUniformDesign.dat",false);

    // Dummy functions with a loop to slow down the function
    double max=-1000000;
    for(int i=0; i<test.getNPatterns(); i++)
    {
        double val = test.getValue("ystar",i);
        max = ((val>=max) ? val : max);
    }
    y[0] = max * x[0];

}
```

In this function the dataset is always the same (`flowrateUniformDesign.dat`) and the multiplier is the only input attribute. The maximum of the dataset is found by creating a `TDataServer` object, by calling the `fileDataRead` method to read the dataset and by looping over the events (this is not at all the best way to do it, but it is a toy model to show what problems can arise when a `TDataServer` object is created in an evaluator use in multi-thread approach). Few important points in this function to prevent from race condition (not thread-safe behaviour):

- the `TDataServer` object is created statistically so that at the end of the function it is automatically destroyed;
- the line below is used to tell the `TDataServer` object not to write his data tree in the ROOT internal list and not to dump it in the archive file when the object is destroyed

  ```
  test.keepFinalTuple(false);
  ```

- the line below is used to tell the `fileDataRead` method not to create the archive ROOT file once the dataset is read (thanks to the optional boolean set to false here)

  ```
  test.fileDataRead("flowrateUniformDesign.dat",false);
  ```

The main function starts then with a block that allows to test the main point here: the method develop by ROOT to prevent the internal list to store object which would lead to race conditions. This block is commented to explain how to run the use-case macro discuss here. The important part is, if one wants to run the macro properly, to call

```
ROOT::EnableThreadSafety();
```

Once this is done, then the macro can be briefly described in the few key steps

1. create the input (multiplier) and output (MultMean) attribute;
2. create the interface to the `multiply` function;
3. create the interface for the runner;

4. create the `TDataServer` object that would contain the multiplier and the results. This object is created within the `onMaster()` part because otherwise there would have been as many dataserver object as there are threads.

5. create a sampler and a design-of-experiments to read 24 times the given dataset;

6. run the computations

At the end, once the macro is launched by using the command below, the

```
root -b -q relauncherCJitFunctionThreadTest.C
```

XIV.8.3.3 Console

```
Processing relauncherCJitFunctionThreadTest.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

************************************************
*    Row   * pouet__n_ * multiplie * MultMean. *
************************************************
*        0 *         1 * 8.1562084 * 1343.9800 *
*        1 *         0 * 4.1412811 * 682.40030 *
*        2 *         2 * 9.4179988 * 1551.8978 *
*        3 *         3 * 6.5650956 * 1081.7964 *
*        4 *         4 * 7.0461633 * 1161.0667 *
*        5 *         5 * 4.7243584 * 778.47978 *
*        6 *         6 * 2.5976785 * 428.04546 *
*        7 *         8 * 5.3607876 * 883.35059 *
*        8 *         7 * 7.4193739 * 1222.5644 *
*        9 *         9 * 5.6692661 * 934.18168 *
*       10 *        10 * 1.0476136 * 172.62578 *
*       11 *        11 * 3.5864637 * 590.97748 *
*       12 *        12 * 1.5680924 * 258.39027 *
*       13 *        13 * 4.8760831 * 803.48098 *
*       14 *        14 * 6.0024502 * 989.08374 *
*       15 *        15 * 9.0195417 * 1486.2400 *
*       16 *        16 * 9.9852106 * 1645.3630 *
*       17 *        17 * 3.1396825 * 517.35689 *
*       18 *        18 * 2.4617970 * 405.65491 *
*       19 *        19 * 6.7059225 * 1105.0019 *
*       20 *        20 * 3.9170611 * 645.45333 *
*       21 *        21 * 7.9536348 * 1310.5999 *
*       22 *        22 * 8.6398795 * 1423.6793 *
*       23 *        23 * 2.1125191 * 348.10091 *
************************************************
```

XIV.8.4 Macro "`relauncherCodeFlowrateSequential.C`"

XIV.8.4.1 Objective

The goal of this macro is to show how to handle a code with a sequential runner. The `flowrate` code is provided with Uranie and has been also used and discussed throughout these macros.

XIV.8.4.2 Macro

```
{

  // Create the TDataServer
  TDataServer *tds = new TDataServer("foo","test");
  tds->fileDataRead("flowrateUniformDesign.dat");

  // Get the attributes
  TAttribute *rw = tds->getAttribute("rw");
  TAttribute *r = tds->getAttribute("r");
  TAttribute *tu = tds->getAttribute("tu");
  TAttribute *tl = tds->getAttribute("tl");
  TAttribute *hu = tds->getAttribute("hu");
  TAttribute *hl = tds->getAttribute("hl");
  TAttribute *l = tds->getAttribute("l");
  TAttribute *kw = tds->getAttribute("kw");

  // Create the output attribute
  TAttribute *yhat = new TAttribute("yhat");
  TAttribute *d = new TAttribute("d");

  // Set the reference input file and the key for each input attributes
  TFlatScript fin("flowrate_input_with_values_rows.in");
  fin.setInputs(8, rw, r, tu, tl, hu, hl, l, kw);

  // The output file of the code
  TFlatResult fout("_output_flowrate_withRow_.dat");
  fout.setOutputs(2, yhat, d);// Passing the attributes to the output file

  // Constructing the code
  TCodeEval mycode( "flowrate -s -r" );
  mycode.setOldTmpDir();
  mycode.addInputFile(&fin); // Adding the input file
  mycode.addOutputFile(&fout); // Adding the output file

  // Create the sequential runner
  TSequentialRun run(&mycode);
  run.startSlave(); //Start the master (necessary even for a sequential)
  if (run.onMaster())
  {
      TLauncher2 lanceur(tds, &run);

      // resolution
      lanceur.solverLoop();
      run.stopSlave(); // Stop the slaves (necessary even for a sequential)
  }

  // Draw the result
  TCanvas *can = new TCanvas("pouet","foo",1);
  tds->Draw("yhat:rw","","colZ");


}
```

Here again, a comparison is drawn with the first Relauncher macro (see Section XIV.8.1.2) and only the differences are pointed out. The first obvious one, in the very first steps in defining the dataserver and the attributes, is that there are two output attributes. The second one (called 'd') will not be used here. The second (and only other difference) with respect to the CINT function code, is the assessor creation shown below:

```
// Set the reference input file and the key for each input attributes
```

```
TFlatScript fin("flowrate_input_with_values_rows.in");
fin.setInputs(8, rw, r, tu, tl, hu, hl, l, kw);

// The output file of the code
TFlatResult fout("_output_flowrate_withRow_.dat");
fout.setOutputs(2, yhat, d);// Passing the attributes to the output file

// Constructing the code
TCodeEval mycode( "flowrate -s -r" );
mycode.setOldTmpDir();
mycode.addInputFile(&fin); // Adding the input file
mycode.addOutputFile(&fout); // Adding the output file
```

The first three lines create the input file instance. It is here a `TFlatScript` object which can basically be compared to a DataServer (or Salome-table) format of the Launcher module for its organisation (particularly with vectors and strings) but without the compulsory header: the order in which you introduce the attribute is then of uttermost importance. The second block of lines is creating the output file object from the `TFlatResult` class (the same remark applies to this object).

Finally the assessor itself is created as an instance of the `TCodeEval` class. The only argument is the command to be run, and it needs at least one input and output file. Apart from that, the runner is created and the rest is crystal clear, leading to the following plot.

XIV.8.4.3 Graph

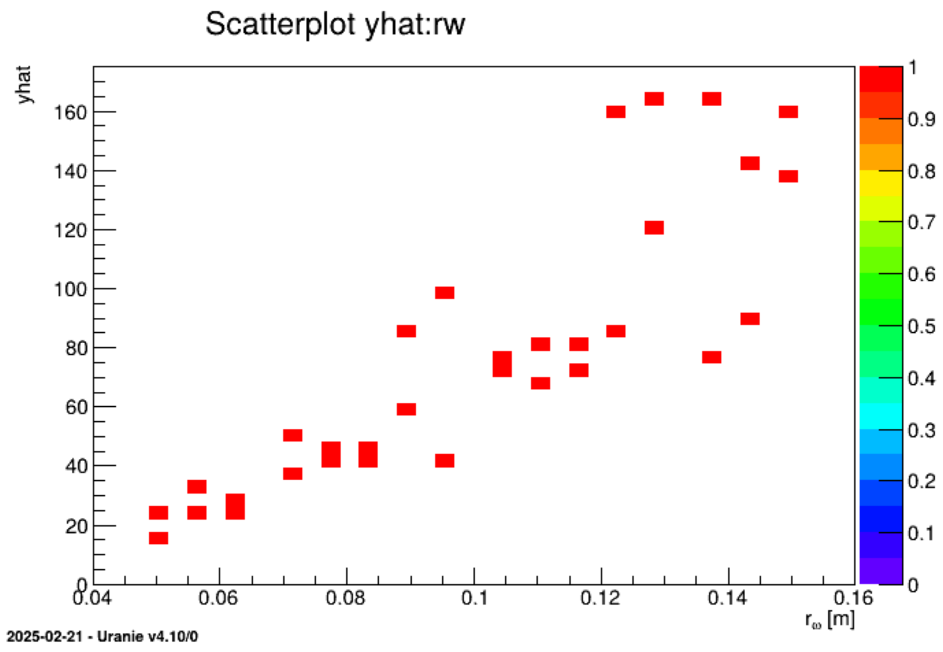


Figure XIV.85: Representation of the output as a function of the first input with a colZ option

XIV.8.5 Macro "relauncherCodeFlowrateSequential_ConstantVar.C"

XIV.8.5.1 Objective

The goal of this macro is to show how to set one of the evaluator's input attribute to a constant value, with a sequential runner. The `flowrate` code is provided with Uranie and has been also used and discussed throughout these macros.

XIV.8.5.2 Macro

```
{
  // Create the TDataServer
  TDataServer *tds = new TDataServer("foo","test");

  // Define the attribute that should be considered as constant
  TAttribute r("r");

  // Add the study attributes ( min, max and nominal values)
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  // The reference input file
  TString sIn = TString("flowrate_input_with_keys.in");

  int nS=15;
  // Generate the Design of Experiments
  TSampling *sampling = new TSampling(tds, "lhs", nS);
  sampling->generateSample();

  // Create the input files
  TKeyScript inputFile( sIn.Data() );
  inputFile.addInput(tds->getAttribute("rw"),"Rw");
  inputFile.addInput(&r,"R"); // Add the constant attribute as an input
  inputFile.addInput(tds->getAttribute("tu"),"Tu");
  inputFile.addInput(tds->getAttribute("tl"),"Tl");
  inputFile.addInput(tds->getAttribute("hu"),"Hu");
  inputFile.addInput(tds->getAttribute("hl"),"Hl");
  inputFile.addInput(tds->getAttribute("l"),"L");
  inputFile.addInput(tds->getAttribute("kw"),"Kw");

  // Create the output attribute
  TAttribute *yhat = new TAttribute("yhat");
  TAttribute *d = new TAttribute("d");

  // Create the output files
  TKeyResult outputFile("_output_flowrate_withKey_.dat");
  outputFile.addOutput(yhat, "yhat");
  outputFile.addOutput(d, "d");

  // Create the user's evaluation function
  TCodeEval eval("flowrate -s -k");
  eval.addInputFile(&inputFile);
  eval.addOutputFile(&outputFile);

  // Create the sequential runner
  TSequentialRun run(&eval);
  run.startSlave(); //Start the master (necessary even for a sequential)
  if (run.onMaster())
  {
      TLauncher2 lanceur(tds, &run);
      // State to the master : r is constant with value 108
      // By default the value is not kept in the tds.
      // The third argument says : yes, keep it for bookkeeping
      lanceur.addConstantValue(&r,108,true);

      // resolution
      lanceur.solverLoop();
      run.stopSlave(); // Stop the slaves (necessary even for a sequential)
  }

  tds->scan("*");


}
```

Here again, a comparison is drawn with the first Relauncher macro (see Section XIV.8.4.2) and only the differences are pointed out. The first obvious one, in the very first steps in defining the dataserver and the attributes, is that instead of reading a database-file, we are generating a design-of-experiments with one big specificity: all the input attributes are properly defined, but **r**.

```
// Define the attribute that should be considered as constant
TAttribute r("r");

// Add the study attributes ( min, max and nominal values)
tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
//  ....
```

A simple design-of-experiments is generated and all the input attributes are provided to the input file of the assessor, event the constant one **r**.

```
// Create the input files
TKeyScript inputFile( sIn.Data() );
inputFile.addInput(tds->getAttribute("rw"),"Rw");
inputFile.addInput(&r,"R"); // Add the constant attribute as an input
inputFile.addInput(tds->getAttribute("tu"),"Tu");
//...
```

The rest is fairly common, up to the `TMaster`-inheriting object specification: the `addConstantValue` method is called to specify that **r** is about to be constant for all ongoing estimation, and it provides it value. The last argument states that the value under consideration should be stored in the ntuple of the dataserver object, as shown in the next section (from the `scan` method).

```
TLauncher2 lanceur(tds, &run);
// State to the master: r is constant with value 108
// By default the value is not kept in the tds.
// The third argument says: yes, keep it for bookkeeping
lanceur.addConstantValue(&r,108,true);
```

XIV.8.5.3 Console

```
*************************************************************************************************************
*    Row   * foo__n *  rw.rw *  tu.tu *  tl.tl *  hu.hu *  hl.hl *    l.l *  kw.kw * yhat.y ↩
     *    d.d *    r.r *
*************************************************************************************************************
*        0 *      0 * 0.1495 * 111790 * 73.820 * 990.90 * 779.83 * 1474.3 * 11220. * 112.01 ↩
     * 3588.9 *    108 *
*        1 *      1 * 0.1394 * 104140 * 95.150 * 1101.5 * 707.21 * 1422.7 * 11493. * 193.62 ↩
     * 6597.5 *    108 *
*        2 *      2 * 0.0557 * 95387. * 84.809 * 1056.2 * 752.94 * 1184.6 * 11967. * 29.880 ↩
     * 330.58 *    108 *
*        3 *      3 * 0.0836 * 74144. * 103.17 * 1051.6 * 819.27 * 1587.4 * 11031. * 35.400 ↩
     * 2431.1 *    108 *
*        4 *      4 * 0.0586 * 65396. * 72.161 * 1003.1 * 710.34 * 1327.5 * 10484. * 24.990 ↩
     *   5757 *    108 *
*        5 *      5 * 0.1203 * 92149. * 65.263 * 1031.6 * 797.34 * 1265.5 * 11638. * 97.386 ↩
     * 1084.8 *    108 *
*        6 *      6 * 0.1319 * 67464. * 93.378 * 1039.4 * 722.54 * 1514.3 * 10996. * 125.33 ↩
     * 2362.4 *    108 *
*        7 *      7 * 0.1059 * 80448. * 112.87 * 1027.1 * 794.32 * 1555.4 *  11846 * 62.403 ↩
     * 1116.3 *    108 *
*        8 *      8 * 0.0784 * 100260 * 105.79 * 1072.7 * 767.70 * 1304.1 * 10152. * 45.867 ↩
     * 521.41 *    108 *
*        9 *      9 * 0.0697 * 105158 * 82.544 * 1020.4 * 726.05 * 1640.3 * 10380. * 28.412 ↩
     * 2802.5 *    108 *
*       10 *     10 * 0.1252 * 89522. * 100.65 * 1006.2 * 742.35 * 1123.9 * 10743. * 123.70 ↩
     * 2676.1 *    108 *
*       11 *     11 * 0.1165 * 73139. * 69.083 * 1108.5 * 809.59 * 1199.0 * 10620. * 112.27 ↩
     * 4991.3 *    108 *
*       12 *     12 * 0.0992 * 86004. * 112.12 * 1069.4 * 763.84 * 1345.2 * 9951.0 * 69.808 ↩
     * 414.71 *    108 *
*       13 *     13 * 0.0718 * 113775 * 90.580 * 1079.1 * 782.95 * 1416.6 * 10076. * 34.135 ↩
     * 1016.8 *    108 *
*       14 *     14 * 0.0902 * 83779. * 80.244 * 1090.6 * 734.33 * 1644.2 *  11443 * 63.236 ↩
     * 2922.3 *    108 *
*************************************************************************************************************
```

XIV.8.6 Macro "relauncherCodeFlowrateThreaded.C"

XIV.8.6.1 Objective

The goal of this macro is to show how to handle a code run on several threads. In order to this, the usual sequential runner will be removed and another runner will be called to do the job. The `flowrate` code is provided with Uranie and has been also used and discussed throughout these macros.

XIV.8.6.2 Macro

```
{

  TAttribute *rw = new TAttribute("rw");
  TAttribute *r = new TAttribute("r");
  TAttribute *tu = new TAttribute("tu");
  TAttribute *tl = new TAttribute("tl");
  TAttribute *hu = new TAttribute("hu");
  TAttribute *hl = new TAttribute("hl");
  TAttribute *l = new TAttribute("l");
  TAttribute *kw = new TAttribute("kw");

  // Create the output attribute
  TAttribute *yhat = new TAttribute("yhat");
  TAttribute *d = new TAttribute("d");

  // Set the reference input file and the key for each input attributes
  TFlatScript fin("flowrate_input_with_values_rows.in");
  fin.setInputs(8, rw, r, tu, tl, hu, hl, l, kw);

  // The output file of the code
  TFlatResult fout("_output_flowrate_withRow_.dat");
  fout.setOutputs(2, yhat, d);// Passing the attributes to the output file

  // Constructing the code
  TCodeEval mycode( "flowrate -s -r" );
  mycode.setOldTmpDir();
  mycode.addInputFile(&fin); // Adding the input file
  mycode.addOutputFile(&fout); // Adding the output file

  // Fix the number of threads
  int nthread = 3;
  // Create the Threaded runner
  TThreadedRun run(&mycode, nthread);
  run.startSlave(); // Start the master
  if (run.onMaster())
  {
      // Create the TDataServer
      TDataServer *tds = new TDataServer("foo","test");
      mycode.addAllInputs(tds);
      tds->fileDataRead("flowrateUniformDesign.dat", kFALSE, kTRUE);

      TLauncher2 lanceur(tds, &run);

      // resolution
      lanceur.solverLoop();
      run.stopSlave(); // Stop the slaves (necessary even for a sequential)

      // Draw the result
      TCanvas *can = new TCanvas("pouet","foo",1);
      tds->Draw("yhat:rw","","colZ");   }

}
```

The only difference when comparing this macro to the previous one (see Section XIV.8.4.2) is the runner creation:

```
// Fix the number of threads
int nthread = 3;
// Create the Threaded runner
TThreadedRun run(&mycode, nthread);
```

The `TSequentialRun` object becomes a `TThreadedRun` object whose construction request on top of the assessor, the number of threads to be used. Apart from that, the master is created and the rest is crystal clear, leading to the following plot.

XIV.8.6.3 Graph

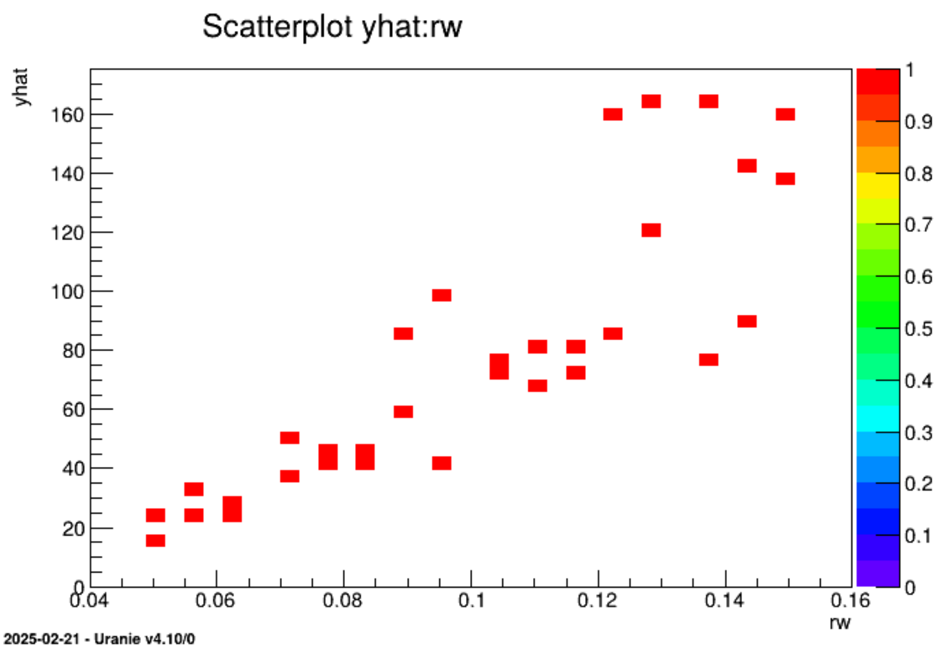


Figure XIV.86: Representation of the output as a function of the first input with a colZ option

XIV.8.7 Macro "relauncherCodeFlowrateMPI.C"

XIV.8.7.1 Objective

The goal of this macro is to show how to handle a code run on several threads with another memory paradigm: when the `TThreadedRun` instance is relying on shared memory (leading to possible thread-safe problem, as discussed in Section VIII.4.2), the MPI implementation is based on the separation of the memory. The communication is made through messages. In order to this, the usual sequential runner will be removed and another runner will be called to do the job. The `flowrate` code is provided with Uranie and has been also used and discussed throughout these macros.

XIV.8.7.2 Macro

```
using namespace URANIE::DataServer;
using namespace URANIE::Relauncher;
using namespace URANIE::MpiRelauncher;

void relauncherCodeFlowrateMPI()
{

    TAttribute *rw = new TAttribute("rw");
    TAttribute *r = new TAttribute("r");
    TAttribute *tu = new TAttribute("tu");
    TAttribute *tl = new TAttribute("tl");
    TAttribute *hu = new TAttribute("hu");
    TAttribute *hl = new TAttribute("hl");
    TAttribute *l = new TAttribute("l");
    TAttribute *kw = new TAttribute("kw");
    // Create the output attribute
    TAttribute *yhat = new TAttribute("yhat");
    TAttribute *d = new TAttribute("d");

    // Set the reference input file and the key for each input attributes
    TFlatScript fin("flowrate_input_with_values_rows.in");
    fin.setInputs(8, rw, r, tu, tl, hu, hl, l, kw);

    // The output file of the code
    TFlatResult fout("_output_flowrate_withRow_.dat");
    fout.setOutputs(2, yhat, d);

    // Instanciation de mon code
    TCodeEval mycode("flowrate -s -r");
    //mycode.setOldTmpDir();
    mycode.addInputFile(&fin);
    mycode.addOutputFile(&fout);

    // Create the MPI runner
    TMpiRun run(&mycode);
    run.startSlave();
    if (run.onMaster())
    {
  // Define the DataServer
  TDataServer tds("tdsflowrate", "Design of Experiments for Flowrate");
  mycode.addAllInputs(&tds);
  tds.fileDataRead("flowrateUniformDesign.dat", kFALSE, kTRUE);

  TLauncher2 lanceur(&tds, &run);

        // resolution
        lanceur.solverLoop();

        tds.exportData("_output_testFlowrateMPI_.dat");

        run.stopSlave();
    }

    delete rw;
    delete r;
    delete tl;
    delete tu;
    delete hl;
    delete hu;
    delete l;
    delete kw;
    delete yhat;
    delete d;
}
```

Here the first difference when comparing this macro to the previous one (see Section XIV.8.6.2) is the runner creation:

```
// Create the MPI runner
TMpiRun run(&mycode);
```

The `TThreadedRun` object becomes a `TMpiRun` object whose construction only requests a pointer to the assessor. Apart from that, the code is very similar, the only difference being the way to call this macro. It should not be run with the usual command:

```
root -l relauncherCodeFlowrateMPI.C
```

Instead, the command line should start with the `mpirun` command as such:

```
mpirun -np N root -l -b -q relauncherCodeFlowrateMPI.C
```

where the *N* part should be replaced by the number of requested threads. Once run, this macro also leads to the following plots.

XIV.8.7.3 Graph

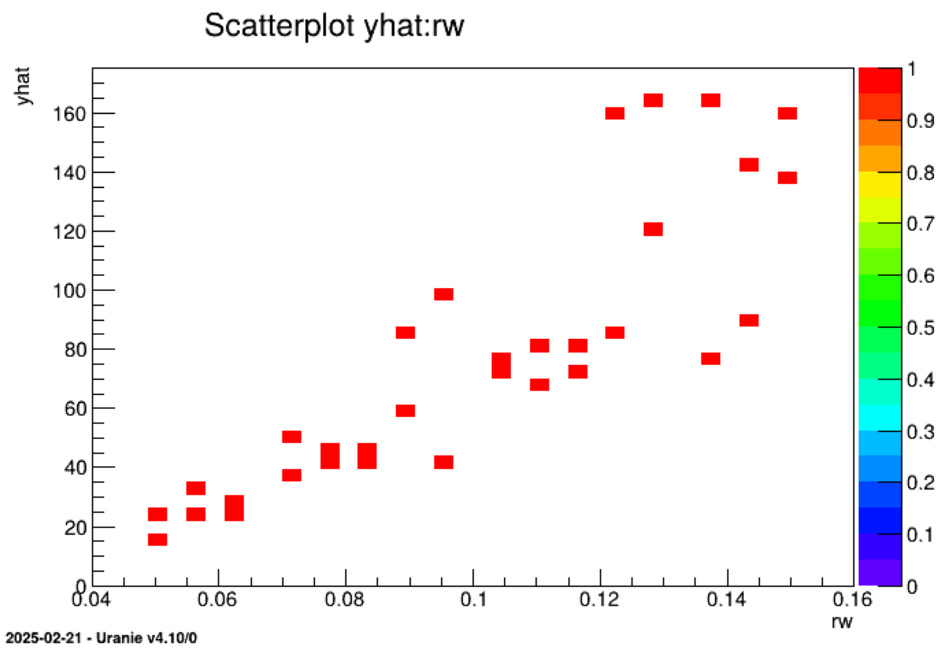


Figure XIV.87: Representation of the output as a function of the first input with a colZ option

XIV.8.8 Macro "`relauncherCodeFlowrateMpiStandalone.C`"

XIV.8.8.1 Objective

The goal of this macro is to show how to handle a code run on several threads with another memory paradigm: when the `TThreadedRun` instance is relying on shared memory (leading to possible thread-safe problem, as discussed in Section VIII.4.2), the MPI implementation is based on the separation of the memory. The communication is made through messages. In order to this, the usual sequential runner will be removed and another runner will be called to do the job. The `flowrate` code is provided with Uranie and has been also used and discussed throughout these macros.

Warning
This macro is different from the one discussed previously in Section XIV.8.7 as here, one wants to handle MPI through standalone compilation (so C++ without using the ROOT interface)

XIV.8.8.2 Macro

```
#include "TAttribute.h"
#include "TCodeEval.h"
#include "TFlatScript.h"
#include "TDataServer.h"
#include "TFlatResult.h"
#include "TMpiRun.h"
#include "TLauncher2.h"

void usage(const char *scmd)
{
    printf("\n");
    printf("Usage: %s initType Type \n", scmd);
    printf("\n");
    printf("  Specific method to test standalone MPI distribution with reoptimizer dummy ↩
        case.\n");
    printf("\t   initType     : The MPI initialisation chosen (explicit <-> 0 or implicit  ↩
        <-> 1) \n");
    printf("\n");
}

int main(int argc, char **argv)
{

    int initType=0; // explicit <-> 0 or implicit <-> 1
    if(argc>1)
    {
        for (int iarg=1; iarg<argc; iarg++)
        {
            if( string(argv[iarg]) == "initType") { initType=atof(argv[iarg+1]); iarg++;}
            else
            {
                cout<<"================================================================"<< ↩
                    endl;
                cout<<"Don't know the following option ? "<<string(argv[iarg])<<endl;
                usage("mpirun -np N ./${EXECNAME}");
                cout<<"================================================================"<< ↩
                    endl;
                return 0;
            }
        }
    }

    // variables
    URANIE::DataServer::TAttribute *rw = new URANIE::DataServer::TAttribute("rw");
    URANIE::DataServer::TAttribute *r = new URANIE::DataServer::TAttribute("r");
    URANIE::DataServer::TAttribute *tu = new URANIE::DataServer::TAttribute("tu");
    URANIE::DataServer::TAttribute *tl = new URANIE::DataServer::TAttribute("tl");
    URANIE::DataServer::TAttribute *hu = new URANIE::DataServer::TAttribute("hu");
    URANIE::DataServer::TAttribute *hl = new URANIE::DataServer::TAttribute("hl");
    URANIE::DataServer::TAttribute *l = new URANIE::DataServer::TAttribute("l");
    URANIE::DataServer::TAttribute *kw = new URANIE::DataServer::TAttribute("kw");

    // Create the output attribute
    URANIE::DataServer::TAttribute *yhat = new URANIE::DataServer::TAttribute("yhat");
    URANIE::DataServer::TAttribute *d = new URANIE::DataServer::TAttribute("d");

    // Set the reference input file and the key for each input attributes
    URANIE::Relauncher::TFlatScript fin("flowrate_input_with_values_rows.in");
    fin.setInputs(8, rw, r, tu, tl, hu, hl, l, kw);
    // The output file of the code
    URANIE::Relauncher::TFlatResult fout("_output_flowrate_withRow_.dat");
    fout.setOutputs(2, yhat, d);

    // Instanciation de mon code
    URANIE::Relauncher::TCodeEval mycode("flowrate -s -r");
    //mycode.setOldTmpDir();
    mycode.addInputFile(&fin);
    mycode.addOutputFile(&fout);

    // Create a runner
    URANIE::MpiRelauncher::TMpiRun *run=NULL;
    if(initType==0)
    {
        MPI_Init(&argc, &argv);
        run = new URANIE::MpiRelauncher::TMpiRun(&mycode);
    }
    else if(initType==1)
        run = new URANIE::MpiRelauncher::TMpiRun(&mycode, &argc, &argv);

    run->startSlave();
    if(run->onMaster())
    {
        // Create the TDS
        URANIE::DataServer::TDataServer tds("launchFlowrate", "launching flowrate with mpi" ↩
            );
        mycode.addAllInputs(&tds);
  tds.fileDataRead("flowrateUniformDesign.dat", kFALSE, kTRUE);

  URANIE::Relauncher::TLauncher2 lanceur(&tds, run);

        // resolution
        lanceur.solverLoop();

        std::stringstream outname; outname<<"_output_testFlowrateMPIStandalone_"<<initType ↩
            <<"_.dat";
        tds.exportData( outname.str().c_str() );

        run->stopSlave();
    }

    delete run;

    return 0;
}
```

Here there are few differences the most obvious one being the proper C++ structure:

- there are several includes that has to be included on top the macro in order to provided the needed headers;
- the main function has to follow a classical form `int main(int argc, char **argv)` which allows to modify parameters on the fly once compiled (as for the `initType` integer here through a dedicated loop). The consequence of this signature will be discussed later on;
- a `usage` function has been written in order to provide interactive guidelines on how to deal with this code.

Apart from this, from the variable definition to the assessor description, the structure is similar to the one discussed previously (see Section XIV.8.6.2). Once this point is reached there are two ways to construct the runner:

- with the classical constructor. In this case, as one is not running a ROOT session, the constructor will not found the interactive command parameters and the `MPI_Init` will not automatically be called, thus meaning that one needs to call this method with the proper parameters as done below:

```
MPI_Init(&argc, &argv);
run = new URANIE::MpiRelauncher::TMpiRun(&mycode);
```

- with the new constructor provided with v4.6.0 whose argument are the usual assessor and the interactive parameters:

```
run = new URANIE::MpiRelauncher::TMpiRun(&mycode, &argc, &argv);
```

The final difference is the fact that this code has to be properly compiled, using MPI-complient compilor in order to be able to produce an executable that will be run later on. This can be done like this:

```
mpicc -o relaunStand relauncherCodeFlowrateMpiStandalone.C `echo $URANIECPPFLAG  ↩
    $URANIELDFLAG` -lstdc++
```

where `mpicc` is the c++ mpi-complient compilor, the argument after `-o` is the executable name, then one can see the input file while the rest is provinding all compiling and linking flag in order to reach the goal. Finally, instead of using the classical `root -l` command, one will use this line

```
mpirun -np N ./relaunStand initType XX
```

where the *N* part should be replaced by the number of requested threads (always strickly greater than 1) and XX should be replaced either by 0 or 1 depending on whether one wants to use the classical `TMpiRun` constructor. Once run, this macro also leads to the following plots (both initialisation are shown).

XIV.8.8.3 Graph

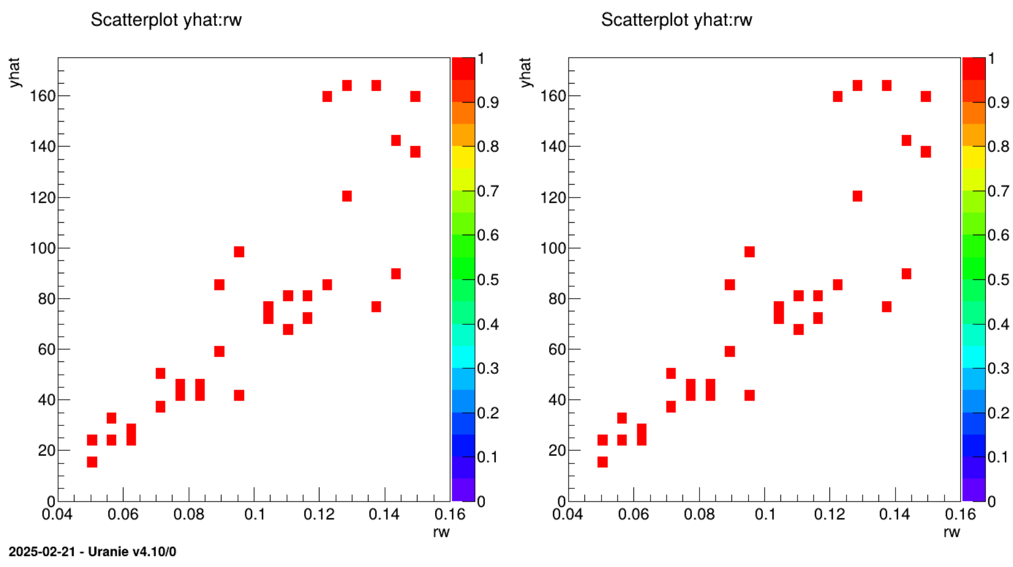


Figure XIV.88: Representation of the output as a function of the first input with a colZ option when using either the classical or dedicated constructor

XIV.8.9 Macro "relauncherCodeFlowrateSequentialFailure.C"

XIV.8.9.1 Objective

The goal of this macro is to show how to handle when a code is returning an error status. Up to version v4.5.0, the input configuration was simply discarded while from any version now, there are discarded but they can be retrieved and store in a dedicated `TDataServer` object. The code used here is the usual `flowrate` model which has been modified to return a non zero exit status without producing an output file.

XIV.8.9.2 Macro

```
{
  // Create the TDataServer
  TDataServer *tds = new TDataServer("foo","test");
  tds->fileDataRead("flowrateUniformDesign.dat");

  // Get the attributes
  TAttribute *rw = tds->getAttribute("rw");
  TAttribute *r = tds->getAttribute("r");
  TAttribute *tu = tds->getAttribute("tu");
  TAttribute *tl = tds->getAttribute("tl");
  TAttribute *hu = tds->getAttribute("hu");
  TAttribute *hl = tds->getAttribute("hl");
  TAttribute *l = tds->getAttribute("l");
  TAttribute *kw = tds->getAttribute("kw");

  // Create the output attribute
  TAttribute *yhat = new TAttribute("yhat");
  TAttribute *d = new TAttribute("d");

  // Set the reference input file and the key for each input attributes
  TFlatScript fin("flowrate_input_with_values_rows.in");
  fin.setInputs(8, rw, r, tu, tl, hu, hl, l, kw);

  // The output file of the code
  TFlatResult fout("_output_flowrate_withRow_.dat");
  fout.setOutputs(2, yhat, d);// Passing the attributes to the output file

  // Constructing the code
  TCodeEval mycode( "flowrate -s -rf" );
  mycode.addInputFile(&fin); // Adding the input file
  mycode.addOutputFile(&fout); // Adding the output file

  // Create the sequential runner
  TSequentialRun run(&mycode);
  run.startSlave(); //Start the master (necessary even for a sequential)

  if (run.onMaster())
  {
      TLauncher2 lanceur(tds, &run);

      // Store the wrong calculation
      TDataServer error("WrongComputations","pouet");
      lanceur.setSaveError(&error);

      // resolution
      lanceur.solverLoop();
      run.stopSlave(); // Stop the slaves (necessary even for a sequential)
      // dump all wrong configurations
      error.getTuple()->SetScanField(-1);
      error.scan("*");
  }

  // Draw the result
  TCanvas *can = new TCanvas("pouet","foo",1);
  tds->Draw("hu:hl");


}
```

Here there are very few differences with the one already introduced in Section XIV.8.4.2. The first one is obviously the command line which is called using "-rf" argument, the f being introduced for failure.

```
TCodeEval mycode( "flowrate -s -rf" );
```

The second difference is the creation of the failure dataserver object in which all wrong configurations will be stored. Once created, it is simply passed to the launcher object through the dedicated method `setSaveError`:

```
// Store the wrong calculation
TDataServer error("WrongComputations","pouet");
lanceur.setSaveError(&error)
```

Once done the code is run and two things are looked at: the fact that in a peculiar area of the input space there are no data anymore (by construction, as shown in Figure XIV.89) and the fact that all configurations are now stored in a dedicated `TDataServer` object which one can dump on screen with the command line below to obtain the second part of the console output seen in Section XIV.8.9.4

```
// dump all wrong configurations
error.getTuple()->SetScanField(-1);
error.scan("*");
```

The first part of the console output shown in Section XIV.8.9.4 is a perfect illustration of the way the relauncher module is discussion failure: the first part is stating that a non-zero return value has been detected

```
Command cd ${RUNNINGDIR}/URA_XXXXXX ; flowrate -s -rf has returned non-zero exit code (255) ↩
    .
 If any different from 127 (usually for unknown command) and 139 (usually for SIGSEV), the  ↩
     exit code meaning is "command" dependent.
```

The second part is letting the user know that no output file has been found (a second reason to consider this configuration as a failure).

```
Cannot open :: ${RUNNINGDIR}/URA_XXXXXX/_output_flowrate_withRow_.dat
```

This pattern is repeated every time a configuration is wrong.

XIV.8.9.3 Graph

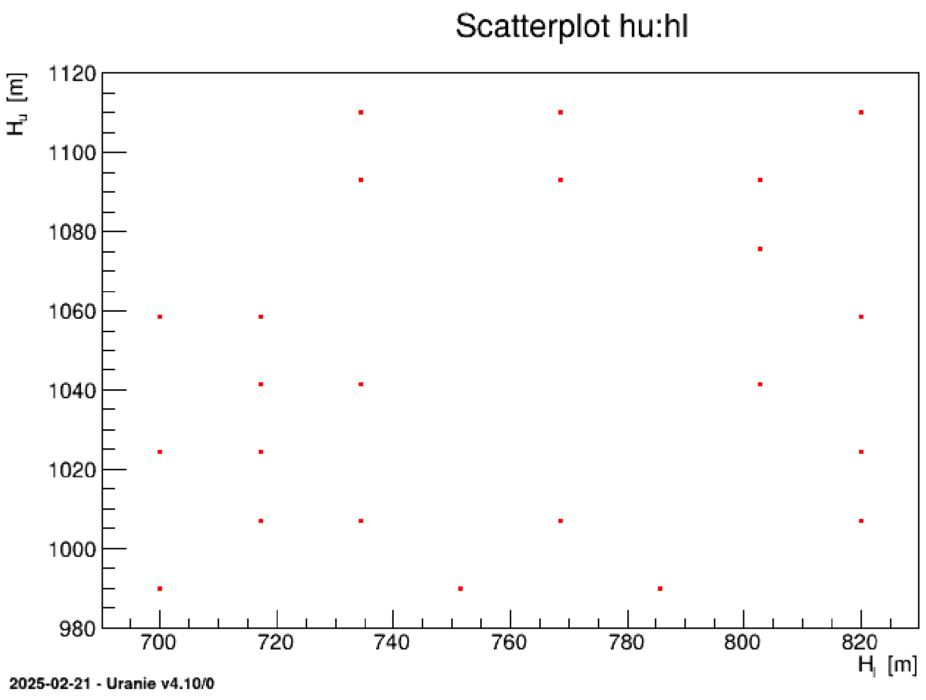


Figure XIV.89: Representation of the output data point when the code is asked to fail on purpose.

XIV.8.9.4 Console

```
Processing relauncherCodeFlowrateSequentialFailure.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

Command cd ${RUNNINGDIR}/URA_XXXXXX ; flowrate -s -rf has returned non-zero exit code (255) ↩
    .
  If any different from 127 (usually for unknown command) and 139 (usually for SIGSEV), the ↩
       exit code meaning is "command" dependent.
Cannot open :: ${RUNNINGDIR}/URA_XXXXXX/_output_flowrate_withRow_.dat
Command cd ${RUNNINGDIR}/URA_XXXXXX ; flowrate -s -rf has returned non-zero exit code (255) ↩
    .
  If any different from 127 (usually for unknown command) and 139 (usually for SIGSEV), the ↩
       exit code meaning is "command" dependent.
Cannot open :: ${RUNNINGDIR}/URA_XXXXXX/_output_flowrate_withRow_.dat
Command cd ${RUNNINGDIR}/URA_XXXXXX ; flowrate -s -rf has returned non-zero exit code (255) ↩
    .
  If any different from 127 (usually for unknown command) and 139 (usually for SIGSEV), the ↩
       exit code meaning is "command" dependent.
Cannot open :: ${RUNNINGDIR}/URA_XXXXXX/_output_flowrate_withRow_.dat
Command cd ${RUNNINGDIR}/URA_XXXXXX ; flowrate -s -rf has returned non-zero exit code (255) ↩
    .
  If any different from 127 (usually for unknown command) and 139 (usually for SIGSEV), the ↩
       exit code meaning is "command" dependent.
Cannot open :: ${RUNNINGDIR}/URA_XXXXXX/_output_flowrate_withRow_.dat
```

```
Command cd ${RUNNINGDIR}/URA_XXXXXX ; flowrate -s -rf has returned non-zero exit code (255) ↩
    .
  If any different from 127 (usually for unknown command) and 139 (usually for SIGSEV), the ↩
       exit code meaning is "command" dependent.
Cannot open :: ${RUNNINGDIR}/URA_XXXXXX/_output_flowrate_withRow_.dat
****************************************************************************************************

*    Row    * WrongComp *     rw.rw *       r.r *     tu.tu *     tl.tl *     hu.hu *     hl ↩
    .hl *       l.l *     kw.kw * ystar.yst *
****************************************************************************************************

*        0 *         4 *    0.0633 *       100 *    115600 *     80.73 *   1075.71 *      ↩
    751.43 *      1600 *  11106.43 *     28.33 *
*        1 *         5 *    0.0633 *  16733.33 *     80580 *     80.73 *   1058.57 *      ↩
    785.71 *      1680 *     12045 *      24.6 *
*        2 *         8 *    0.0767 *       100 *    115600 *     80.73 *   1075.71 *      ↩
    751.43 *      1520 *  10793.57 *     42.44 *
*        3 *        12 *      0.09 *  16733.33 *     63070 *       116 *   1075.71 *      ↩
    751.43 *      1120 *  11419.29 *     83.77 *
*        4 *        23 *    0.1233 *  16733.33 *     63070 *      63.1 *   1041.43 *      ↩
    785.71 *      1680 *     12045 *     86.73 *
****************************************************************************************************
```

XIV.8.10 Macro "relauncherCodeMultiTypeKey.C"

XIV.8.10.1 Objective

The objective of this macro is to test the case where vectors and strings are produced as outputs, using the code described in Section XIV.4.22.1, with a Key format, obtained by doing:

```
multitype -mtKey
```

The resulting output file, named `_output_multitype_mt_Key_.dat` looks like:

```
w1 = nine
v1 = -0.512095
v1 = 0.039669
v1 = -1.3834
v1 = 1.37667
v1 = 0.220672
v1 = 0.633267
v1 = 1.37027
v1 = -0.765636
v2 = 14.1981
v2 = 14.0855
v2 = 10.7848
v2 = 9.45476
v2 = 9.17308
v2 = 6.60804
v2 = 10.0711
v2 = 14.1761
v2 = 10.318
v2 = 12.5095
v2 = 15.6614
v2 = 10.3452
v2 = 9.41101
v2 = 7.47887
f1 = 32.2723
```

```
w2 = eight
```

XIV.8.10.2 Macro Uranie

```
{

  // Create the TDataServer and create the seed attribute
  TDataServer *tds = new TDataServer("foo", "multitype usecase");
  tds->addAttribute( new TUniformDistribution("seed",0,100000));

  //Create DOE
  TSampling *tsam = new TSampling(tds,"lhs",100);
  tsam->generateSample();

  // Create output attribute pointers
  TAttribute *w1 = new TAttribute("w1", TAttribute::kString);
  TAttribute *w2 = new TAttribute("w2", TAttribute::kString);
  TAttribute *v1 = new TAttribute("v1", TAttribute::kVector);
  TAttribute *v2 = new TAttribute("v2", TAttribute::kVector);
  TAttribute *f1 = new TAttribute("f1");

  // Create the input files
  TFlatScript inputFile("multitype_input.dat");
  inputFile.setInputs(1, tds->getAttribute("seed"), "seed");

  // Create the output files
  TKeyResult outputFile("_output_multitype_mt_Key_.dat");
  outputFile.addOutput(w1, "w1");
  outputFile.addOutput(v1, "v1");
  outputFile.addOutput(v2, "v2");
  outputFile.addOutput(f1, "f1");
  outputFile.addOutput(w2, "w2");

  // Create the user's evaluation function
  TCodeEval eval("multitype -mtKey");
  eval.addInputFile(&inputFile); // Add the input file
  eval.addOutputFile(&outputFile); // Add the output file

  //Create the runner
  TSequentialRun runner(&eval);

  // Start the slaves
  runner.startSlave();
  if (runner.onMaster())
  {

      // Create the launcher
      TLauncher2 lanceur(tds, &runner);
      lanceur.solverLoop();

      // Stop the slave processes
      runner.stopSlave();

  }

  //Produce control plot
  TCanvas *Can = new TCanvas("Can","Can",10,10,1000,800);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw(); pad->cd();
  tds->drawPairs("w1:v1:v2:f1:w2");
}
```

The beginning of the code is pretty common to many other macros: creating a dataserver and input attributes (here the only one is the seed, needed for the random generator to produce vectors and strings). A sampling object is created as well to produce a 100-points design-of-experiments and the output attributes are created, as such:

```
// Create output attribute pointers
TAttribute *w1 = new TAttribute("w1", TAttribute::kString);
TAttribute *w2 = new TAttribute("w2", TAttribute::kString);
TAttribute *v1 = new TAttribute("v1", TAttribute::kVector);
TAttribute *v2 = new TAttribute("v2", TAttribute::kVector);
TAttribute *f1 = new TAttribute("f1");
```

This is where the specificity of the vector and string is precised. It will be passed on to the rest of the code automatically. The rest is common to many relauncher job (for instance Section XIV.8.4) with the only difference being that the output file is a key type one. It results in the following plots.

XIV.8.10.3 Graph

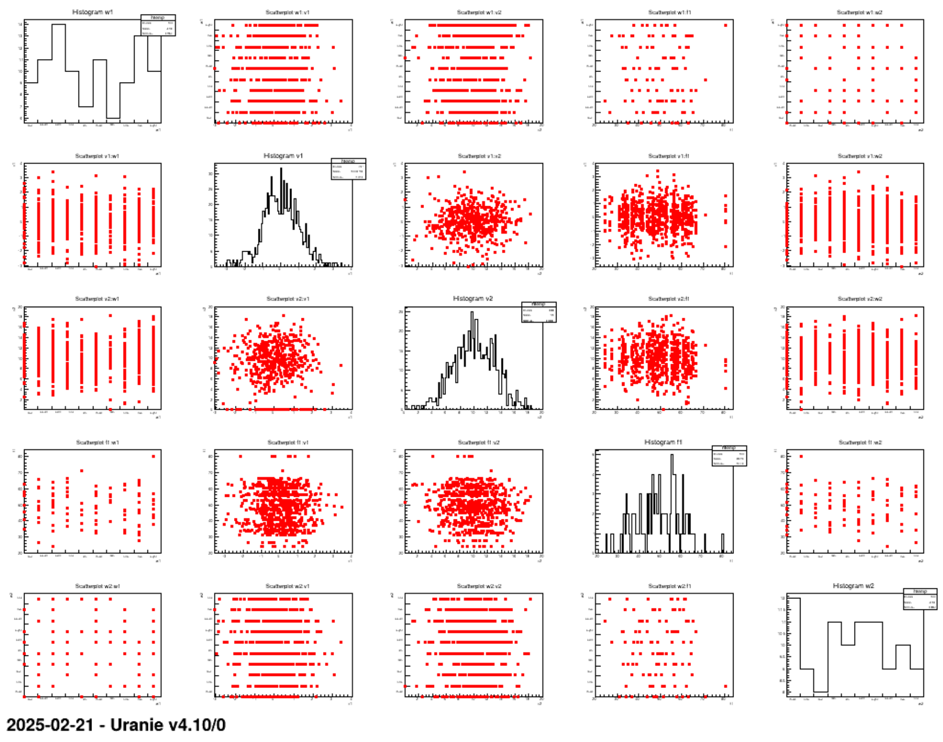


Figure XIV.90: Graph of the macro "relauncherCodeMultiTypeKey.C"

XIV.8.11 Macro "relauncherCodeMultiTypeKeyEmptyVectors.C"

XIV.8.11.1 Objective

The objective of this macro is to test the case where vectors and strings are produced as outputs, using the code described in Section XIV.4.22.1, with a Key format, obtained by doing:

```
multitype -mtKey -empty
```

Unlike what's done to in Section XIV.8.10, the *"-empty"* allows the code to generate empty vectors and not only vectors whose size would be between 1 and 15 elements. The resulting output file used is a key-format one in a condensate form, named `_output_multitype_mt_Key_condensate_.dat` looks like:

```
w1 = nine
v1 = [ -0.512095,0.039669,-1.3834,1.37667,0.220672,0.633267,1.37027,-0.765636 ]
v2 = [  ↩
    14.1981,14.0855,10.7848,9.45476,9.17308,6.60804,10.0711,14.1761,10.318,12.5095,15.6614,10.3452,9.4110
     ]
f1 = 32.2723
w2 = eight
```

XIV.8.11.2 Macro Uranie

```
{

  // Create the TDataServer and create the seed attribute
  TDataServer *tds = new TDataServer("foo", "multitype usecase");
  tds->addAttribute( new TUniformDistribution("seed",0,100000));

  //Create DOE
  TSampling *tsam = new TSampling(tds,"lhs",100);
  tsam->generateSample();

  // Create output attribute pointers
  TAttribute *w1 = new TAttribute("w1", TAttribute::kString);
  TAttribute *w2 = new TAttribute("w2", TAttribute::kString);
  TAttribute *v1 = new TAttribute("v1", TAttribute::kVector);
  TAttribute *v2 = new TAttribute("v2", TAttribute::kVector);
  TAttribute *f1 = new TAttribute("f1");

  // Create the input files
  TFlatScript inputFile("multitype_input.dat");
  inputFile.setInputs(1, tds->getAttribute("seed"), "seed");

  // Create the output files
  TKeyResult outputFile("_output_multitype_mt_Key_condensate_.dat");
  outputFile.addOutput(w1, "w1");
  outputFile.addOutput(v1, "v1");
  outputFile.addOutput(v2, "v2");
  outputFile.addOutput(f1, "f1");
  outputFile.addOutput(w2, "w2");
  outputFile.setVectorProperties("[",",","]");

  // Create the user's evaluation function
  TCodeEval eval("multitype -mtKey -empty");
  eval.addInputFile(&inputFile); // Add the input file
  eval.addOutputFile(&outputFile); // Add the output file

  //Create the runner
  TSequentialRun runner(&eval);

  // Start the slaves
  runner.startSlave();
  if (runner.onMaster())
  {

      // Create the launcher
      TLauncher2 lanceur(tds, &runner);
      lanceur.solverLoop();
      // Stop the slave processes
      runner.stopSlave();

  }

  //Produce control plot
  TCanvas *Can = new TCanvas("Can","Can",10,10,1000,800);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw(); pad->Divide(1,2);
  pad->cd(1);
  tds->getTuple()->SetLineColor(2); tds->getTuple()->SetLineWidth(2);
  tds->Draw("size__v1");
  pad->cd(2);
  tds->Draw("size__v2");

}
```

The beginning of the code is pretty common to the macro already discussed in Section XIV.8.10.2. Apart from the command difference discussed in the objective above through the *"-empty"* argument, the main difference with previous macro is the way the output file is declared. Despite from changing the name, the vector properties are set by calling the `setVectorProperties` method to emphasize how to read the information.

```
// Create the output files
TKeyResult outputFile("_output_multitype_mt_Key_condensate_.dat");
outputFile.addOutput(w1, "w1");
outputFile.addOutput(v1, "v1");
outputFile.addOutput(v2, "v2");
outputFile.addOutput(f1, "f1");
outputFile.addOutput(w2, "w2");
outputFile.setVectorProperties("[",",","]");
```

Apart from this, the code is smooth and the final results one can be interested in the size of the vectors produced when empty vectors are allowed. This is produced though the following lines, and the resulting plots are shown in Figure XIV.91.

```
//Produce control plot
TCanvas *Can = new TCanvas("Can","Can",10,10,1000,800);
TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw(); pad->Divide(1,2);
pad->cd(1);
tds->getTuple()->SetLineColor(2); tds->getTuple()->SetLineWidth(2);
tds->Draw("size__v1");
pad->cd(2);
tds->Draw("size__v2");
```

If the output file was not properly formatted, then one can have issues with this specific case (empty vectors). The consequences are shown in Section XIV.8.12.

XIV.8.11.3 Graph

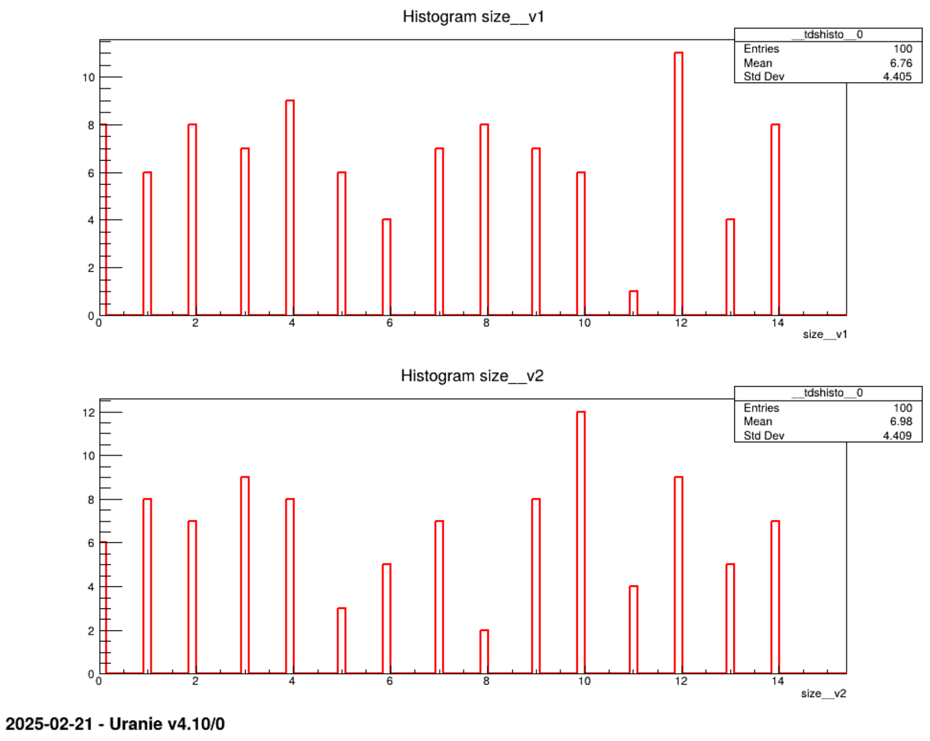


Figure XIV.91: Graph of the macro "`relauncherCodeMultiTypeKeyEmptyVectors.C`"

XIV.8.12 Macro "`relauncherCodeMultiTypeKeyEmptyVectorsAsFailure.C`"

XIV.8.12.1 Objective

The objective of this macro is to test the case where vectors and strings are produced as outputs, using the code described in Section XIV.4.22.1, with a Key format, obtained by doing:

```
multitype -mtKey -empty
```

Unlike what's done to in Section XIV.8.10, the *"-empty"* allows the code to generate empty vectors and not only vectors whose size would be between 1 and 15 elements. The resulting output file used is a key-format in a very row form, meaning that evey new element of the vectors are written as a new key-line. This file, named `_output_multitype_mt_Key_.dat` could looks like this:

```
w1 = nine
v1 = -0.512095
v1 = 0.039669
v1 = -1.3834
v1 = 1.37667
v1 = 0.220672
v1 = 0.633267
v1 = 1.37027
v1 = -0.765636
v2 = 14.1981
v2 = 14.0855
v2 = 10.7848
v2 = 9.45476
```

```
v2 = 9.17308
v2 = 6.60804
v2 = 10.0711
v2 = 14.1761
v2 = 10.318
v2 = 12.5095
v2 = 15.6614
v2 = 10.3452
v2 = 9.41101
v2 = 7.47887
f1 = 32.2723
w2 = eight
```

XIV.8.12.2 Macro Uranie

```
{
  // Create the TDataServer and create the seed attribute
  TDataServer *tds = new TDataServer("foo", "multitype usecase");
  tds->addAttribute( new TUniformDistribution("seed",0,100000));

  //Create DOE
  TSampling *tsam = new TSampling(tds,"lhs",100);
  tsam->generateSample();

  // Create output attribute pointers
  TAttribute *w1 = new TAttribute("w1", TAttribute::kString);
  TAttribute *w2 = new TAttribute("w2", TAttribute::kString);
  TAttribute *v1 = new TAttribute("v1", TAttribute::kVector);
  TAttribute *v2 = new TAttribute("v2", TAttribute::kVector);
  TAttribute *f1 = new TAttribute("f1");

  // Create the input files
  TFlatScript inputFile("multitype_input.dat");
  inputFile.setInputs(1, tds->getAttribute("seed"), "seed");

  // Create the output files
  TKeyResult outputFile("_output_multitype_mt_Key_.dat");
  outputFile.addOutput(w1, "w1");
  outputFile.addOutput(v1, "v1");
  outputFile.addOutput(v2, "v2");
  outputFile.addOutput(f1, "f1");
  outputFile.addOutput(w2, "w2");

  // Create the user's evaluation function
  TCodeEval eval("multitype -mtKey -empty");
  eval.addInputFile(&inputFile); // Add the input file
  eval.addOutputFile(&outputFile); // Add the output file

  //Create the runner
  TSequentialRun runner(&eval);

  // Start the slaves
  runner.startSlave();
  if (runner.onMaster())
  {

      // Create the launcher
      TLauncher2 lanceur(tds, &runner);

      // Store the wrong calculation
      TDataServer error("WrongComputations","pouet");
      lanceur.setSaveError(&error);

      lanceur.solverLoop();

      // dump all wrong configurations
      cout<<"\nFailed configurations: "<<endl;
      error.getTuple()->SetScanField(-1);
      error.scan("*");

      // Stop the slave processes
      runner.stopSlave();

  }
  //Produce control plot
  TCanvas *Can = new TCanvas("Can","Can",10,10,1000,800);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw(); pad->cd();
  tds->drawPairs("w1:v1:v2:f1:w2");

}
```

The beginning of the code is pretty common to the macro already discussed in Section XIV.8.10.2. Apart from the command difference discussed in the objective above through the *"-empty"* argument, the main difference with previous macro is the failure dataserver declaration and the output console that would be discussed later-on. The former is done through the following lines:

```
// Store the wrong calculation
TDataServer error("WrongComputations","pouet");
lanceur.setSaveError(&error);
```

Once the code is run, the configuration leading to empty vectors are gathered in the failure dataserver and dumped on screen through the following lines:

```
// dump all wrong configurations
error.getTuple()->SetScanField(-1);
error.scan("*");
```

The final part is the way to represent the results: as for the use-case macro discussed in Section XIV.8.10, all data are plotted in a pair plot and this is summarised in Figure XIV.92. From this picture one should really pay attention to the number of entries to spot that some configuration are missing. Luckily when looking at the console in Section XIV.8.12.4. This time (unlike the failure in Section XIV.8.9) the code is returning a zero output status (because the code actually worked fine) but as from time to time one the two vectors is empty, no entry is written in the output whose format is too simple (as it consist only in dumping vector elements by elements) this is why the only message is the fact that, from time to time, one vector information is missing.

XIV.8.12.3 Graph

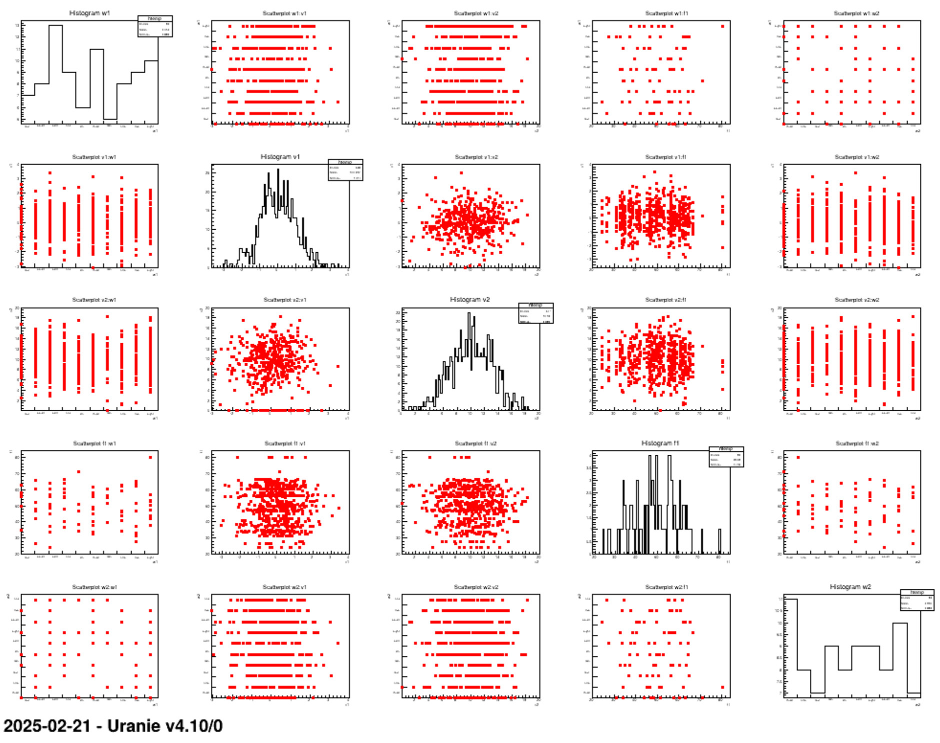


Figure XIV.92: Graph of the macro "relauncherCodeMultiTypeKeyEmptyVectorsAsFailure.C"

XIV.8.12.4 Console

```
Processing relauncherMultiTypeKeyEmptyVectorsAsFailure.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

TKeyResult(_output_multitype_mt_Key_.dat): v1 Not found
TKeyResult(_output_multitype_mt_Key_.dat): v2 Not found
TKeyResult(_output_multitype_mt_Key_.dat): v1 Not found
TKeyResult(_output_multitype_mt_Key_.dat): v2 Not found
TKeyResult(_output_multitype_mt_Key_.dat): v1 Not found
TKeyResult(_output_multitype_mt_Key_.dat): v2 Not found
TKeyResult(_output_multitype_mt_Key_.dat): v1 Not found
TKeyResult(_output_multitype_mt_Key_.dat): v1 Not found
TKeyResult(_output_multitype_mt_Key_.dat): v2 Not found
TKeyResult(_output_multitype_mt_Key_.dat): v2 Not found
TKeyResult(_output_multitype_mt_Key_.dat): v1 Not found
TKeyResult(_output_multitype_mt_Key_.dat): v1 Not found
TKeyResult(_output_multitype_mt_Key_.dat): v1 Not found
TKeyResult(_output_multitype_mt_Key_.dat): v2 Not found

Failed configurations:
************************************
*    Row   * WrongComp * seed.seed *
************************************
```

```
*        0 *         9 *  93759.29 *
*        1 *        33 * 74051.957 *
*        2 *        35 * 71909.957 *
*        3 *        50 * 4183.9188 *
*        4 *        54 * 41806.234 *
*        5 *        57 * 28298.703 *
*        6 *        66 * 64903.722 *
*        7 *        69 * 47690.947 *
*        8 *        73 * 89415.222 *
*        9 *        79 * 30656.411 *
*       10 *        84 * 31627.094 *
*       11 *        86 * 63698.481 *
*       12 *        89 * 13461.926 *
*       13 *        99 * 52994.014 *
************************************
```

XIV.8.13 Macro "relauncherCodeReadMultiType.C"

XIV.8.13.1 Objective

The objective of this macro is to test the case where vectors and strings are used as inputs, using the code described in Section XIV.4.22.2, with a Key format, obtained by doing:

```
multitype -ReadmtKey
```

The input values will be read from a database which is produced with the `multitype -mt` code, as no sampling is available yet to produce vectors and strings. The database file is `readmultitype_sampling.dat` which looks like this:

```
#NAME: foo
#TITLE: TDS for flowrate
#DATE: Mon Oct  3 23:50:34 2016
#COLUMN_NAMES: v1| w1| v2| w2| f1| foo__n__iter__
#COLUMN_TYPES: V|S|V|S|D|D

-6.901933299378e-02,-1.292435959913e-01,4.558876683004e-01,5.638486368789e ↩
    -01,-4.767582766745e-02,7.102109543136e-03,2.819049677902e-01,-2.019788081790e ↩
    +00,-2.604401028584e+00,-1.617682380292e+00,2.894560949798e-02,-3.493905850261e-01 six  ↩
    1.142449011404e+01,7.318948216271e+00,1.502260859231e+01,6.041193793062e ↩
    +00,6.729445145907e+00,1.128096968597e+01 zero 3.425632316777e+01 0.000000000000e+00
-6.923200061823e-01,-4.798721931875e-01,-1.329893204384e+00,1.292933726829e+00 zero  ↩
    1.249911290435e+01,6.309239169117e+00,1.596653626442e+01,5.500878012739e ↩
    +00,1.322535550082e+01,7.070984389647e+00,1.708574150702e+00,1.265915339220e+01 two  ↩
    4.295175025115e+01 1.000000000000e+00
5.773813268848e-01,-3.512405673973e-01,-6.870089014992e-01,1.273074555211e-01 nine  ↩
    1.242682578759e+01,1.109680842701e+01,1.670410641828e+01,7.296321908492e ↩
    +00,8.732800753443e+00,1.262906549132e+01,8.882310687564e+00,1.104280818003e+01 five  ↩
    5.591437936893e+01 2.000000000000e+00
5.518508915499e-01,2.438158138873e-01,1.111784497742e+00,-1.517566514667e+00,7.146879916125 ↩
    e-01,2.328439269321e+00,-1.251913839951e+00,8.876684186954e-01,-1.383023165632e ↩
    +00,-8.192089693621e-01,-1.079524713568e-01,6.595650273375e-01,-2.275345802432e ↩
    -03,1.304354557600e+00 nine 1.021975159505e+01,4.995433740783e+00,1.108628156181e ↩
    +01,1.041110604995e+01,1.111365770153e+01,6.365695806343e+00,6.374053973239e ↩
    +00,6.854423942510e+00,7.144262333164e+00 two 4.093776591421e+01 3.000000000000e+00
2.403942476958e-01,6.868091212609e-01,-1.561012830108e+00,1.937806684989e ↩
    +00,-1.465851888061e+00,5.367279844359e-02,-1.263005327899e+00,-1.132259472701e+00 two  ↩
    7.382048319627e+00,5.874867917970e+00,1.158191378461e+01,1.073321314846e+01 six  ↩
    6.980549752305e+01 4.000000000000e+00
```

```
2.220485143391e+00,-5.787212569267e-01,8.843648237689e-01,2.020662891124e+00,1.066403357312 ↩
    e+00,-5.817432767992e-01,3.063023900800e-01,-7.393588637933e-01 two 2.049656723853e ↩
    +00,9.679003878866e+00,7.338089623518e+00,1.235630702472e+01,1.509238505697e ↩
    +01,1.034077492413e+01,1.116077550501e+01,7.179221834787e+00,1.582041236432e ↩
    +01,9.204085091129e+00,4.707490792498e+00,1.618155764288e+01 five 3.507773555061e+01  ↩
    5.000000000000e+00
8.908373817765e-01,-2.446355046704e-01,-1.900125532005e+00 seven 1.351254851860e ↩
    +01,9.297087139459e+00,1.130966904782e+01,1.219245848701e+01,1.012996566249e ↩
    +01,7.150071600452e+00,1.097549218518e+01,1.443074761657e+01 five 4.464560504112e+01  ↩
    6.000000000000e+00
-2.514644600888e+00,1.633579305804e+00 one 1.229098312451e+01,1.013486836958e ↩
    +01,1.243386772880e+01,1.071783135260e+01,1.453735777922e+01,7.995593455015e ↩
    +00,9.753966962919e+00,5.924583770352e+00,6.187713988125e+00,1.061975242996e ↩
    +01,6.650425922126e+00 four 4.553396475968e+01 7.000000000000e+00
-1.347811599520e+00,-1.259450135534e+00,1.812553405758e+00 five 7.717018655412e ↩
    +00,1.053283796180e+01,7.404059210327e+00 eight 6.695868880279e+01 8.000000000000e+00
-1.258360863204e-01,-9.000566818602e-01,7.039146852797e-01,1.015917277706e ↩
    +00,-2.397650482929e-01 four 4.346717386417e+00,1.033024889324e+01,7.183787459050e ↩
    +00,8.742095837835e+00,1.277095440277e+01,8.685683828779e+00,9.321006265935e ↩
    +00,6.353438157123e+00,8.552570119034e+00 six 4.381313066586e+01 9.000000000000e+00
```

For every pattern, an input file is created with the Key condensate format, as the other key format is not practical (and usable). This input file looks like this:

```
w1 = nine
v1 = [ -0.512095,0.039669,-1.3834,1.37667,0.220672,0.633267,1.37027,-0.765636 ]
v2 = [  ↩
    14.1981,14.0855,10.7848,9.45476,9.17308,6.60804,10.0711,14.1761,10.318,12.5095,15.6614,10.3452,9
     ]
f1 = 32.2723
w2 = eight
```

The resulting output file, named `_output_multitype_readmt_Key_.dat` looks like:

```
thev1 = -0.2397650482929
thev2 = 9.321006265935
```

XIV.8.13.2 Macro Uranie

```
{
  //inputs
  TDataServer tds("foo","TDS for flowrate");
  tds.fileDataRead("readmultitype_sampling.dat");

  // Input attribute
  TAttribute *w1 = tds.getAttribute("w1");
  TAttribute *w2 = tds.getAttribute("w2");
  TAttribute *v1 = tds.getAttribute("v1");
  TAttribute *v2 = tds.getAttribute("v2");
  TAttribute *f1 = tds.getAttribute("f1");

  // output attribute
  TAttribute *thev1 = new TAttribute("thev1");
  TAttribute *thev2 = new TAttribute("thev2");
  gSystem->Exec("multitype -mtKey");
   // Create the output files
  TKeyScript inputFile("_output_multitype_mt_Key_condensate_.dat");
  inputFile.setInputs(5, w1, "w1", v1, "v1", v2, "v2", f1, "f1", w2, "w2");
  // Create the output files
  TKeyResult outputFile("_output_multitype_readmt_Key_.dat");
  outputFile.addOutput(thev1, "thev1");
  outputFile.addOutput(thev2, "thev2");

  // Create the user's evaluation function
  TCodeEval eval("multitype -ReadmtKey");
  eval.addInputFile(&inputFile);
  eval.addOutputFile(&outputFile);

  TSequentialRun runner(&eval);
  //TThreadedRun runner(&eval,2);

  runner.startSlave();
  if (runner.onMaster())
    {

      // Create the launcher
      TLauncher2 lanceur(&tds, &runner);
      lanceur.solverLoop();

      // Stop the slave processes
      runner.stopSlave();

    }

  tds.Scan("thev1:thev2");

}
```

The code is pretty straightforward, the fact that input attributes are vectors and strings is explained in the input file `readmultitype_sampling.dat`. One line is added to be sure that an example of input file is present (the file `_output_multitype_mt_Key_condensate_.dat`) by calling:

```
gSystem->Exec("multitype -mtKey");
```

The rest is very common and a screenshot of the result displayed in console is provided in the following subsection.

XIV.8.13.3 Console

```
Processing relauncherCodeReadMultiType.C...
************************************
*    Row   *     thev1 *     thev2 *
************************************
*        0 * 0.2819049 * 11.424490 *
*        1 * -0.692320 * 15.966536 *
*        2 * -12345678 * 12.629065 *
*        3 * -0.819208 * 11.086281 *
*        4 * -1.561012 * -12345678 *
*        5 * 0.8843648 * 10.340774 *
*        6 * -12345678 * 7.1500716 *
*        7 * 1.6335793 * 14.537357 *
*        8 * -12345678 * -12345678 *
*        9 * -0.239765 * 9.3210062 *
************************************
```

XIV.8.14 Macro "relauncherComposeMultitypeAndReadMultiType.C"

XIV.8.14.1 Objective

The objective of this macro is to to combine two different assessor in a chain, so that output attributes of the first assessor is the input attributes of the second one. This example combined the `multitype` code to produce vectors and strings as outputs (as explained in Section XIV.4.22.1) and use these vectors and strings as inputs, using the code described in Section XIV.4.22.2.

XIV.8.14.2 Macro Uranie

```
{
  //inputs
  TDataServer tds("foo","TDS for multitype");
  tds.fileDataRead("multitype_sampling.dat");

  //output attributes...
  // ... for code 1
  TAttribute *w1 = new TAttribute("w1", URANIE::DataServer::TAttribute::kString);
  TAttribute *w2 = new TAttribute("w2", URANIE::DataServer::TAttribute::kString);
  TAttribute *v1 = new TAttribute("v1", URANIE::DataServer::TAttribute::kVector);
  TAttribute *v2 = new TAttribute("v2", URANIE::DataServer::TAttribute::kVector);
  TAttribute *f1 = new TAttribute("f1");

  // ... for code 2
  TAttribute *thev1 = new TAttribute("thev1");
  TAttribute *thev2 = new TAttribute("thev2");

  // ==================================================================
  // ========================== Code 1 ================================
  // ==================================================================

  // Create the input files
  TFlatScript inputFile1("multitype_input.dat");
  inputFile1.setInputs(1, tds.getAttribute("seed"), "seed");

  // Create the output files
  TKeyResult outputFile1("_output_multitype_mt_Key_.dat");
  outputFile1.addOutput(w1, "w1");
  outputFile1.addOutput(v1, "v1");
  outputFile1.addOutput(v2, "v2");
  outputFile1.addOutput(f1, "f1");
  outputFile1.addOutput(w2, "w2");

  // Create the user's evaluation function
  TCodeEval eval1("multitype -mtKey");
  eval1.addInputFile(&inputFile1);
  eval1.addOutputFile(&outputFile1);

  // ==================================================================
  // ========================== Code 2 ================================
  // ==================================================================

  // Create the output files
  TKeyScript inputFile2("_output_multitype_mt_Key_condensate_.dat");
  inputFile2.setInputs(5, w1, "w1", v1, "v1", v2, "v2", f1, "f1", w2, "w2");

  // Create the output files
  TKeyResult outputFile2("_output_multitype_readmt_Key_.dat");
  outputFile2.addOutput(thev1, "thev1");
  outputFile2.addOutput(thev2, "thev2");

  // Create the user's evaluation function
  TCodeEval eval2("multitype -ReadmtKey");
  eval2.addInputFile(&inputFile2);
  eval2.addOutputFile(&outputFile2);

  // ==================================================================
  // ======================= Composition ==============================
  // ==================================================================

  // Create the composition
  TComposeEval eval;
  // Add the code one-by-one, in the right order
  eval.addEval(&eval1);
  eval.addEval(&eval2);

  // Create the runner by providing the TComposeEval
  TSequentialRun runner(&eval);

  runner.startSlave();
  if (runner.onMaster())
  {

    // Create the launcher
    TLauncher2 lanceur(&tds, &runner);
    lanceur.solverLoop();

    // Stop the slave processes
    runner.stopSlave();

    tds.exportData("pouet.dat");

  }

  tds.Scan("thev1:thev2");

}
```

The code looks very much as the one in two previous examples. First a sample of 10 seed values are read from an input file. Then, output attributes are defined for the first code, as in Section XIV.8.10.2.

```
TAttribute *w1 = new TAttribute("w1", URANIE::DataServer::TAttribute::kString);
TAttribute *w2 = new TAttribute("w2", URANIE::DataServer::TAttribute::kString);
TAttribute *v1 = new TAttribute("v1", URANIE::DataServer::TAttribute::kVector);
TAttribute *v2 = new TAttribute("v2", URANIE::DataServer::TAttribute::kVector);
TAttribute *f1 = new TAttribute("f1");
```

The output attributes are defined for the first code, as in in Section XIV.8.13.2.

```
TAttribute *thev1 = new TAttribute("thev1");
TAttribute *thev2 = new TAttribute("thev2");
```

The assessor are then defined with input and output files and the composition is finally done: it is an assessor in in which we store the other assessors that should be run, in the correct order, as follows:

```
TComposeEval eval;
// Add the code one-by-one, in the right order
eval.addEval(&eval1);
eval.addEval(&eval2);
```

The rest is very common and a screenshot of the result displayed in console is provided in the following subsection.

XIV.8.14.3 Console

```
Processing relauncherComposeMultitypeAndReadMultiType.C...
************************************
*    Row   *      thev1 *      thev2 *
************************************
*        0 * 0.2819049 * 11.424490 *
*        1 * -0.692320 * 15.966536 *
*        2 * -12345678 * 12.629065 *
*        3 * -0.819208 * 11.086281 *
*        4 * -1.561012 * -12345678 *
*        5 * 0.8843648 * 10.340774 *
*        6 * -12345678 * 7.1500716 *
*        7 * 1.6335793 * 14.537357 *
*        8 * -12345678 * -12345678 *
*        9 * -0.239765 * 9.3210062 *
************************************
```

XIV.8.15 Macro "relauncherCodeFlowrateSequential_TemporaryVar.C"

XIV.8.15.1 Objective

The goal of this macro is to show how to hide one of the evaluator's attribute and not to store it in the final dataserver. This is considered when a composition is done for instance, in which many variables might be intermediate needed ones, resulting from an assessor and used as input to one of the following, but of no interest to the user at the end. The flowrate code is provided with Uranie and has been also used and discussed throughout these macros.

XIV.8.15.2 Macro

```
void IncreaseD(double *x, double *y)
{
  y[0] = x[0] + 1;
}

void relauncherCodeFlowrateSequential_TemporaryVar()
{
  // Create the TDataServer
  TDataServer *tds = new TDataServer("foo","test");

  // Define the attribute that should be considered as constant
  TAttribute r("r");

  // Add the study attributes ( min, max and nominal values)
  tds->addAttribute( new TUniformDistribution("rw", 0.05, 0.15));
  tds->addAttribute( new TUniformDistribution("tu", 63070.0, 115600.0));
  tds->addAttribute( new TUniformDistribution("tl", 63.1, 116.0));
  tds->addAttribute( new TUniformDistribution("hu", 990.0, 1110.0));
  tds->addAttribute( new TUniformDistribution("hl", 700.0, 820.0));
  tds->addAttribute( new TUniformDistribution("l", 1120.0, 1680.0));
  tds->addAttribute( new TUniformDistribution("kw", 9855.0, 12045.0));

  // The reference input file
  TString sIn = TString("flowrate_input_with_keys.in");
```

```
int nS=15;
// Generate the Design of Experiments
TSampling *sampling = new TSampling(tds, "lhs", nS);
sampling->generateSample();

// Create the input files
TKeyScript inputFile( sIn.Data() );
inputFile.addInput(tds->getAttribute("rw"),"Rw");
inputFile.addInput(&r,"R");
inputFile.addInput(tds->getAttribute("tu"),"Tu");
inputFile.addInput(tds->getAttribute("tl"),"Tl");
inputFile.addInput(tds->getAttribute("hu"),"Hu");
inputFile.addInput(tds->getAttribute("hl"),"Hl");
inputFile.addInput(tds->getAttribute("l"),"L");
inputFile.addInput(tds->getAttribute("kw"),"Kw");

// Create the output attributes
TAttribute *yhat = new TAttribute("yhat");
TAttribute *d = new TAttribute("d");

// Create the output files
TKeyResult outputFile("_output_flowrate_withKey_.dat");
outputFile.addOutput(yhat, "yhat");
outputFile.addOutput(d, "d");

// Create the user's evaluation function
TCodeEval eval1("flowrate -s -k");
eval1.addInputFile(&inputFile);
eval1.addOutputFile(&outputFile);

// Create a second evaluation function that uses d to change it slightly
TAttribute *incd = new TAttribute("incd");
TCIntEval eval2("IncreaseD");
eval2.addInput(d);
eval2.addOutput(incd);

// Create the composition
TComposeEval eval;
// Add the code one-by-one, in the right order
eval.addEval(&eval1);
eval.addEval(&eval2);

// Create the sequential runner
TSequentialRun run(&eval);
run.startSlave(); //Start the master (necessary even for a sequential)
if (run.onMaster())
{
    TLauncher2 lanceur(tds, &run);
    // State to the master : d is an output attribute and I'm interested in its value
    // but I don't want to keep it in the end. It might be usefull for another evaluator
    lanceur.addTemporary(d);
    lanceur.addConstantValue(&r,108);

    // resolution
    lanceur.solverLoop();
    run.stopSlave(); // Stop the slaves (necessary even for a sequential)
}

tds->scan("*");
```

```
}
```

Here again, a comparison is drawn with the macro in which we set an attribute to a constant value (see Section XIV.8.5), so only the differences are pointed out. The very first one is contained in the beginning lines: a new dummy function, so that we can have a composition of two assessors, this function only adding one to the provided parameter.

```
void IncreaseD(double *x, double *y)
{
  y[0] = x[0] + 1;
}
```

The rest is exactly as for Section XIV.8.5, up to the interface with the newy create function:

```
// Create a second evaluation function that uses d to change it slightly
TAttribute *incd = new TAttribute("incd");
TCIntEval eval2("IncreaseD");
eval2.addInput(d);
eval2.addOutput(incd);

// Create the composition
TComposeEval eval;
// Add the code one-by-one, in the right order
eval.addEval(&eval1);
eval.addEval(&eval2);
```

A new output attribute is created, called **incd** for increased **d**, and the dummy function is defined as taking **d** as input and incd as output. Then the composition is done by chaining flowrate with the new dummy function. The rest is fairly common, up to the `TMaster`-inheriting object specification: the `addTemporary` method is called to specify that **d** is read from the output of flowrate and can be pass to the rest of the chain, but it will not be kept in the final dataserver. The `addConstantValue` is also used just changing the final parameters to show that if nothing is specified, then the value of **r** is not stored and this might be tricky for bookkeeping. The results is shown in the next section (from the `scan` method) and can be compared to Section XIV.8.5.3 for consistency check.

```
TLauncher2 lanceur(tds, &run);
// State to the master: d is an output attribute and I'm interested in its value
// but I don't want to keep it in the end. It might be usefull for another evaluator
lanceur.addTemporary(d);
lanceur.addConstantValue(&r,108);
```

XIV.8.15.3 Console

```
*****************************************************************************************************
*    Row   * foo__n *  rw.rw *  tu.tu *  tl.tl *  hu.hu *  hl.hl *    l.l *  kw.kw * yhat.y ↩
     * incd.i *
*****************************************************************************************************
*        0 *      0 * 0.1495 * 111790 * 73.820 * 990.90 * 779.83 * 1474.3 * 11220. * 112.01 ↩
     * 3589.9 *
*        1 *      1 * 0.1394 * 104140 * 95.150 * 1101.5 * 707.21 * 1422.7 * 11493. * 193.62 ↩
     * 6598.5 *
*        2 *      2 * 0.0557 * 95387. * 84.809 * 1056.2 * 752.94 * 1184.6 * 11967. * 29.880 ↩
     * 331.58 *
*        3 *      3 * 0.0836 * 74144. * 103.17 * 1051.6 * 819.27 * 1587.4 * 11031. * 35.400 ↩
     * 2432.1 *
*        4 *      4 * 0.0586 * 65396. * 72.161 * 1003.1 * 710.34 * 1327.5 * 10484. * 24.990 ↩
     *   5758 *
*        5 *       5 * 0.1203 * 92149. * 65.263 * 1031.6 * 797.34 * 1265.5 * 11638. * 97.386 ↩
    * 1085.8 *
*        6 *       6 * 0.1319 * 67464. * 93.378 * 1039.4 * 722.54 * 1514.3 * 10996. * 125.33 ↩
    * 2363.4 *
*        7 *       7 * 0.1059 * 80448. * 112.87 * 1027.1 * 794.32 * 1555.4 *  11846 * 62.403 ↩
    * 1117.3 *
*        8 *       8 * 0.0784 * 100260 * 105.79 * 1072.7 * 767.70 * 1304.1 * 10152. * 45.867 ↩
    * 522.41 *
*        9 *       9 * 0.0697 * 105158 * 82.544 * 1020.4 * 726.05 * 1640.3 * 10380. * 28.412 ↩
    * 2803.5 *
*       10 *      10 * 0.1252 * 89522. * 100.65 * 1006.2 * 742.35 * 1123.9 * 10743. * 123.70 ↩
    * 2677.1 *
*       11 *      11 * 0.1165 * 73139. * 69.083 * 1108.5 * 809.59 * 1199.0 * 10620. * 112.27 ↩
    * 4992.3 *
*       12 *      12 * 0.0992 * 86004. * 112.12 * 1069.4 * 763.84 * 1345.2 * 9951.0 * 69.808 ↩
    * 415.71 *
*       13 *      13 * 0.0718 * 113775 * 90.580 * 1079.1 * 782.95 * 1416.6 * 10076. * 34.135 ↩
    * 1017.8 *
*       14 *      14 * 0.0902 * 83779. * 80.244 * 1090.6 * 734.33 * 1644.2 *  11443 * 63.236 ↩
    * 2923.3 *
********************************************************************************************** ↩
```

XIV.9 Macros Reoptimizer

XIV.9.1 Macro "`reoptimizeHollowBarCode.C`"

XIV.9.1.1 Objective

The objective of the macro is to optimize the section of the hollow bar defined in Section IX.2.2 using the NLopt solvers (reducing it to a single-criterion optimisation as already explained in Section IX.3. This can be done with different solvers, the results being achieved within more or less time and following the requested constraints with more or less accuracy (depending on the hypothesis embedded by the chosen solver).

XIV.9.1.2 Macro Uranie

```
{

    using namespace URANIE::DataServer;
    using namespace URANIE::Relauncher;
    using namespace URANIE::Reoptimizer;

    // variables
    TAttribute x("x", 0.0, 1.0),
      y("y", 0.0, 1.0),
      thick("thick"), // thickness
      sect("sect"), // section of the pipe
      dist("dist"); // distortion

    // Creating the TCodeEval, dumping output of the dummy python in an output file
    string python_exec = "python3";
    if(string(gSystem->GetBuildArch()) == "win64")
        python_exec.pop_back();
    TCodeEval code((python_exec +" bar.py > bartoto.dat").data());
    // Pass the python script itself as a input file. x and y will be modified in bar.py ↩
        directly
    TKeyScript inputfile("bar.py");
    inputfile.addInput(&x,"x");
    inputfile.addInput(&y,"y");
    code.addInputFile(&inputfile);

    // precise the name of the output file in which to read the three output variables
    TFlatResult outputfile("bartoto.dat");
    outputfile.addOutput(&thick);
    outputfile.addOutput(&sect);
    outputfile.addOutput(&dist);
    code.addOutputFile(&outputfile);

    // Create a runner
    TSequentialRun runner(&code);
    runner.startSlave(); // Usual Relauncher construction

    if(runner.onMaster())
    {
        // Create the TDS
        TDataServer tds("vizirDemo", "Param de l'opt vizir pour la barre");
        tds.addAttribute(&x);
        tds.addAttribute(&y);

        // Choose a solver
        TNloptCobyla solv;
        //TNloptBobyqa solv;
        //TNloptPraxis solv;
        //TNloptNelderMead solv;
        ///TNloptSubplexe solv;

        // Create the single-objective constrained optimizer
        TNlopt opt(&tds, &runner, &solv);

        // add the objective
        opt.addObjective(&sect); // minimizing the section

  // and the constrains
        TLesserFit constrDist(14);
        opt.addConstraint(&dist,&constrDist); // on the distortion (dist < 14)
        TGreaterFit positiv(0.4);
        opt.addConstraint(&thick,&positiv); // and on the thickness (thick > 0.4)

        // Starting point and maximum evaluation
        vector<double> point{0.9 , 0.2};
        opt.setStartingPoint(point.size(),&point[0]);
        opt.setMaximumEval(1000);

        opt.solverLoop(); // running the optimization

        // Stop the slave processes
        runner.stopSlave();

        // solution
        tds.getTuple()->Scan("*","","colsize=9 col=:::5:4");
    }


}
```

The variables are defined as follow:

```
// variables
TAttribute x("x", 0.0, 1.0),
y("y", 0.0, 1.0),
thick("thick"), // thickness
sect("sect"), // section of the pipe
dist("dist"); // distortion
```

where the first two are the input ones while the last ones are computed using the provided code (as explained in Section IX.2.2). This code is configured through these lines:

```
// Creating the TCodeEval, dumping output of the dummy python in an output file
TCodeEval code("python bar.py > bartoto.dat");

// Pass the python script itself as a input file. x and y will be modified in bar.py ↩
    directly
TKeyScript inputfile("bar.py");
inputfile.addInput(&x,"x");
inputfile.addInput(&y,"y");
code.addInputFile(&inputfile);

// precise the name of the output file in which to read the three output variables
TFlatResult outputfile("bartoto.dat");
outputfile.addOutput(&thick);
outputfile.addOutput(&sect);
outputfile.addOutput(&dist);
code.addOutputFile(&outputfile);
```

The usual Relauncher construction is followed, using a `TSequentialRun` runner and the solver is chosen in these lines:

```
// Choose a solver
TNloptCobyla solv;
//TNloptBobyqa solv;
//TNloptPraxis solv;
//TNloptNelderMead solv;
///TNloptSubplexe solv;
```

Combining the runner, solver and dataserver, the master object is created and the objective and constraint are defined (keeping in mind that only single-criterion problems are implemented when dealing with NLopt, so the distortion criteria is downgraded to a constraint). This is done in

```
// Create the single-objective constrained optimizer
TNlopt opt(&tds, &runner, &solv);

// add the objective
opt.addObjective(&sect); // minimizing the section

// and the constrains
TLesserFit constrDist(14);
opt.addConstraint(&dist,&constrDist); // on the distortion (dist < 14)
TGreaterFit positiv(0.4);
opt.addConstraint(&thick,&positiv); // and on the thickness (thick > 0.4)
```

Finally the starting point is set along with the maximal number of evaluation just before starting the loop.

```
// Starting point and maximum evaluation
vector<double> point{0.9 , 0.2};
opt.setStartingPoint(point.size(),&point[0]);
opt.setMaximumEval(1000);
opt.solverLoop(); // running the optimisation
```

XIV.9.1.3 Console

This macro leads to the following result

```
Processing reoptimizeHollowBarCode.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

|....:....|....:....|....:....|....:....
|***************************************************************************
*    Row   * vizirDemo *       x.x *       y.y * thick * sect * dist.dist *
***************************************************************************
*        0 *         0 * 0.5173156 * 0.1173173 * 0.399 * 0.25 * 13.999986 *
***************************************************************************
```

XIV.9.2 Macro "reoptimizeHollowBarCodeMultiStart.C"

XIV.9.2.1 Objective

The objective of the macro is to optimize the section of the hollow bar defined in Section IX.2.2 using the NLopt solvers (reducing it to a single-criterion optimisation as already explained in Section IX.3. It is largely based on the previous macro, the main change being the fact that we allow different starting points.

XIV.9.2.2 Macro Uranie

```
{

    using namespace URANIE::DataServer;
    using namespace URANIE::Relauncher;
    using namespace URANIE::Reoptimizer;

    // variables
    TAttribute x("x", 0.0, 1.0),
      y("y", 0.0, 1.0),
      thick("thick"), // thickness
      sect("sect"), // section of the pipe
      dist("dist"); // distortion

    // Creating the TCodeEval, dumping output of the dummy python in an output file
    string python_exec = "python3";
    if(string(gSystem->GetBuildArch()) == "win64")
        python_exec.pop_back();
    TCodeEval code((python_exec + " bar.py > bartoto.dat").data());

    // Pass the python script itself as a input file. x and y will be modified in bar.py ↩
        directly
    TKeyScript inputfile("bar.py");
    inputfile.addInput(&x,"x");
    inputfile.addInput(&y,"y");
    code.addInputFile(&inputfile);

    // precise the name of the output file in which to read the three output variables
    TFlatResult outputfile("bartoto.dat");
    outputfile.addOutput(&thick);
    outputfile.addOutput(&sect);
    outputfile.addOutput(&dist);
    code.addOutputFile(&outputfile);

    // Create a runner
    TSequentialRun runner(&code);
    runner.startSlave(); // Usual Relauncher construction

    if(runner.onMaster())
    {
        // Create the TDS
        TDataServer tds("vizirDemo", "Param de l'opt vizir pour la barre");
        tds.addAttribute(&x);
        tds.addAttribute(&y);

        // Choose a solver
        TNloptCobyla solv;
        //TNloptBobyqa solv;
        //TNloptPraxis solv;
        //TNloptNelderMead solv;
        ///TNloptSubplexe solv;

        // Create the single-objective constrained optimizer
        TNlopt opt(&tds, &runner, &solv);

        // add the objective
        opt.addObjective(&sect); // minimizing the section

  // and the constrains
        TLesserFit constrDist(14);
        opt.addConstraint(&dist,&constrDist); // on the distortion (dist < 14)
        TGreaterFit positiv(0.4);
        opt.addConstraint(&thick,&positiv); // and on the thickness (thick > 0.4)

        // Starting points
        vector<double> p1{0.9 , 0.2}, p2{0.7 , 0.1}, p3{0.5 , 0.4};
        opt.setStartingPoint(p1.size(),&p1[0]);
        opt.setStartingPoint(p2.size(),&p2[0]);
        opt.setStartingPoint(p3.size(),&p3[0]);

        // Set maximum evaluation
        opt.setMaximumEval(1000);

        opt.solverLoop(); // running the optimization

        // Stop the slave processes
        runner.stopSlave();

        // solution
        tds.getTuple()->Scan("*","","colsize=9 col=:::5:4");
    }


}
```

As stated previously, the purpose of this macro is to use different starting points for optimisation fully based on the macro shown in Section XIV.9.1. The only difference is highlighted here:

```
// Starting points
vector<double> p1{0.9 , 0.2}, p2{0.7 , 0.1}, p3{0.5 , 0.4};
opt.setStartingPoint(p1.size(),&p1[0]);
opt.setStartingPoint(p2.size(),&p2[0]);
opt.setStartingPoint(p3.size(),&p3[0]);
```

The results of this is that optimisation is performed three times, using the three starting points provided. Here it is done sequentially, but obviously, the main idea is that it is a convenient way to parallelise these optimisation. This could be done for instance, simply by changing the runner line from

```
TSequentialRun runner(&code);
```

to, for instance in our case with 3 starting points

```
TThreadedRun runner(&code,4);
```

XIV.9.2.3 Console

This macro leads to the following result

```
Processing reoptimizeHollowBarCodeMS.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

|....:....|....:....|....:....|....:....|....:....0050
|....:....|....:....
|....:....|....:....|....:....0100
|..
..:....|....:....|....:...
***************************************************************************
*    Row   * vizirDemo *       x.x *       y.y * thick * sect * dist.dist *
***************************************************************************
*        0 *         0 * 0.5173155 * 0.1173213 * 0.399 * 0.25 * 14.000005 *
*        1 *         1 * 0.5173156 * 0.1173173 * 0.399 * 0.25 * 13.999986 *
*        2 *         2 * 0.5173155 * 0.1173155 *   0.4 * 0.25 *        14 *
***************************************************************************
```

XIV.9.3 Macro "reoptimizeHollowBarCodevizir.C"

XIV.9.3.1 Objective

The objective of the macro is to optimize the section and distortion of the hollow bar defined in Section IX.2.2 using the evolutionary solvers. This can be done with different solvers, the one chosen here being the `TVizirGenetic` one.

XIV.9.3.2 Macro Uranie

```
{

    using namespace URANIE::DataServer;
    using namespace URANIE::Relauncher;
    using namespace URANIE::Reoptimizer;

    // variables
    TAttribute x("x", 0.0, 1.0),
      y("y", 0.0, 1.0),
      thick("thick"), // thickness
      sect("sect"), // section of the pipe
      dist("dist"); // distortion

    // Creating the TCodeEval, dumping output of the dummy python in an output file
    string python_exec = "python3";
    if(string(gSystem->GetBuildArch()) == "win64")
        python_exec.pop_back();
    TCodeEval code((python_exec + " bar.py > bartoto.dat").data());

    // Pass the python script itself as a input file. x and y will be modified in bar.py ↩
        directly
    TKeyScript inputfile("bar.py");
    inputfile.addInput(&x,"x");
    inputfile.addInput(&y,"y");
    code.addInputFile(&inputfile);

    // precise the name of the output file in which to read the three output variables
    TFlatResult outputfile("bartoto.dat");
    outputfile.addOutput(&thick);
    outputfile.addOutput(&sect);
    outputfile.addOutput(&dist);
    code.addOutputFile(&outputfile);

    // Create a runner
    TSequentialRun runner(&code);
    runner.startSlave(); // Usual Relauncher construction

    if(runner.onMaster())
    {
        // Create the TDS
        TDataServer tds("vizirDemo", "Param de l'opt vizir pour la barre");
        tds.addAttribute(&x);
        tds.addAttribute(&y);

        // create the vizir genetic solver
        TVizirGenetic solv;
        // Set the size of the population to 150, and a maximum number of evaluation at ↩
            15000
        solv.setSize(200,15000);

        // Create the multi-objective constrained optimizer
        TVizir2 opt(&tds, &runner, &solv);

        // add the objective
        opt.addObjective(&sect); // minimizing the section
        opt.addObjective(&dist); // minimizing the distortion

        // and the constrains
        TGreaterFit positiv(0.4);
        opt.addConstraint(&thick,&positiv);  //on thickness (thick > 0.4)

        opt.solverLoop(); // running the optimization

        // Stop the slave processes
        runner.stopSlave();
```

```
  TCanvas *fig1 = new TCanvas("fig1","Pareto Zone",5,64,1270,667);
  int phi=12;  int theta=30;
  TPad *pad1 = new TPad("pad1","",0,0.03,1,1);
  TPad *pad2 = new TPad("pad2","",0,0.03,1,1);
  pad2->SetFillStyle(4000); //will be transparent

  pad1->Draw(); pad1->Divide(2,1); pad1->cd(1); gPad->SetPhi(phi); gPad->SetTheta(theta);

  gStyle->SetLabelSize(0.03); gStyle->SetLabelSize(0.03,"Y"); gStyle->SetLabelSize(0.03,"Z" ↩
      );

  tds.getTuple()->Draw("sect:y:x");
  //Get the TH3 to change Z axis color
  TH3F *htemp = (TH3F*)gPad->GetPrimitive("htemp");
  htemp->SetTitle("");
  htemp->GetZaxis()->SetLabelColor(2); htemp->GetZaxis()->SetAxisColor(2); htemp->GetZaxis ↩
      ()->SetTitleColor(2);

  fig1->cd();
  pad2->Draw();
  pad2->Divide(2,1); pad2->cd(1); gPad->SetFillStyle(4000); gPad->SetPhi(phi); gPad-> ↩
      SetTheta(theta);
  tds.getTuple()->SetMarkerColor(4);
  tds.getTuple()->Draw("dist:y:x");
  htemp = (TH3F*)gPad->GetPrimitive("htemp");
  htemp->SetTitle("");
  htemp->GetZaxis()->SetLabelColor(4); htemp->GetZaxis()->SetAxisColor(4); htemp->GetZaxis ↩
      ()->SetTitleColor(4);
  htemp->GetZaxis()->SetTickSize( -1*htemp->GetZaxis()->GetTickLength() );
  htemp->GetZaxis()->SetLabelOffset( -15*htemp->GetZaxis()->GetLabelOffset() );
  htemp->GetZaxis()->LabelsOption("d");
  htemp->GetZaxis()->SetTitleOffset( -1.5*htemp->GetZaxis()->GetTitleOffset() );
  htemp->GetZaxis()->RotateTitle( );

  pad2->cd(2);
  tds.getTuple()->SetMarkerColor(2);
  tds.draw("dist:sect");


    }
}
```

The variables are defined as follow:

```
// variables
TAttribute x("x", 0.0, 1.0),
y("y", 0.0, 1.0),
thick("thick"), // thickness
sect("sect"), // section of the pipe
dist("dist"); // distortion
```

where the first two are the input ones while the last ones are computed using the provided code (as explained in Section IX.2.2). This code is configured through these lines

```
// Creating the TCodeEval, dumping output of the dummy python in an output file
TCodeEval code("python bar.py > bartoto.dat");

// Pass the python script itself as a input file. x and y will be modified in bar.py  ↩
    directly
TKeyScript inputfile("bar.py");
inputfile.addInput(&x,"x");
inputfile.addInput(&y,"y");
code.addInputFile(&inputfile);

// precise the name of the output file in which to read the three output variables
TFlatResult outputfile("bartoto.dat");
outputfile.addOutput(&thick);
outputfile.addOutput(&sect);
outputfile.addOutput(&dist);
code.addOutputFile(&outputfile);
```

The usual Relauncher construction is followed, using a `TSequentialRun` runner and the solver is chosen in these lines

```
// create the vizir genetic solver
TVizirGenetic solv;
// Set the size of the population to 150, and a maximum number of evaluation at 15000
solv.setSize(200,15000);
```

Combining the runner, solver and dataserver, the master object is created and the objective and constraint are defined. This is done in:

```
// Create the multi-objective constrained optimizer
TVizir2 opt(&tds, &runner, &solv);

// add the objective
opt.addObjective(&sect); // minimizing the section
opt.addObjective(&dist); // minimizing the distortion

// and the constrains
TGreaterFit positiv(0.4);
opt.addConstraint(&thick,&positiv);  //on thickness (thick > 0.4);
```

Finally the optimisation is launched and the rest of code is providing the graphical result shown in next section.

XIV.9.3.3 Graph

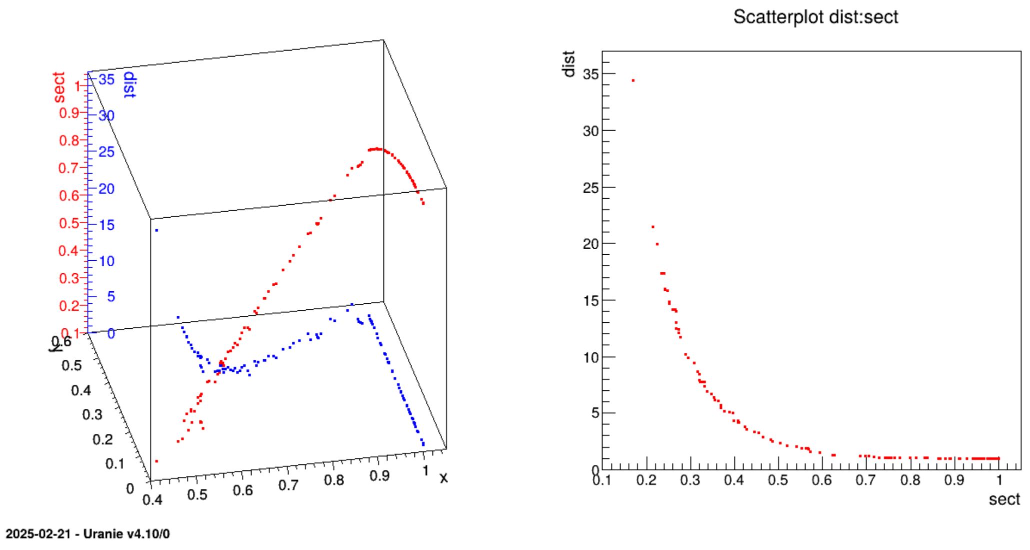


Figure XIV.93: Graph of the macro "`reoptimizeHollowBarCodeVizir.C`"

XIV.9.4 Macro "reoptimizeHollowBarVizirMoead.C"

XIV.9.4.1 Objective

The objective of the macro is to optimize the section and distortion of the hollow bar defined in Section IX.2.2 using the evolutionary solvers, with a reduce number of points to compose the Pareto set/front. This example is comparing both the usual Vizir genetic algorithm and the MOEAD implementation that is meant to be a many-objective criteria algorithm. A short discussion on the many-objective aspect can be found in [30].

XIV.9.4.2 Macro Uranie

```
{
    #define nbPoints 20
    #define total 4*nbPoints

    using namespace URANIE::DataServer;
    using namespace URANIE::Relauncher;
    using namespace URANIE::Reoptimizer;

    // variables
    TAttribute x("x", 0.0, 1.0),
      y("y", 0.0, 1.0),
      thick("thick"), // thickness
      sect("sect"), // section of the pipe
      dist("dist"); // distortion

     gROOT->LoadMacro("UserFunctions.C");

    // Creating the assessor using the analytical function
    TCIntEval code("barAllCost");
    code.addInput(&x);
    code.addInput(&y);
    code.addOutput(&thick);
    code.addOutput(&sect);
    code.addOutput(&dist);

    // Create a runner
    TSequentialRun runner(&code);
    runner.startSlave(); // Usual Relauncher construction

    int nMax=3000;
    if(runner.onMaster())
    {
        // ==================================================
        // ========= Classical Vizir implementation =========
        // ==================================================

        // Create the TDS
        TDataServer tds_viz("vizirDemo", "Vizir parameter dataser");
        tds_viz.addAttribute(&x);
        tds_viz.addAttribute(&y);

        // create the vizir genetic solver
        TVizirGenetic solv_viz;
        // Set the size of the population to 150, and a maximum number of evaluation at ↩
            15000
        solv_viz.setSize(nbPoints,nMax);

        // Create the multi-objective constrained optimizer
        TVizir2 opt_viz(&tds_viz, &runner, &solv_viz);

         // add the objective
        opt_viz.addObjective(&sect); // minimizing the section
        opt_viz.addObjective(&dist); // minimizing the distortion

        // and the constrains
        TGreaterFit positiv(0.4);
        opt_viz.addConstraint(&thick,&positiv);  //on thickness (thick > 0.4)

        opt_viz.solverLoop(); // running the optimization

        // ==================================================
        // ============== MOEAD implementation ==============
        // ==================================================

         // Create the TDS
        TDataServer tds_moead("vizirDemo", "Vizir parameter dataser");
        tds_moead.addAttribute(&x);
        tds_moead.addAttribute(&y);

        // create the vizir genetic solver
        TVizirGenetic solv_moead;
        solv_moead.setMoeadDiversity(nbPoints);
        solv_moead.setStoppingCriteria(1);
        solv_moead.setSize(0, nMax, 200);

        // Create the multi-objective constrained optimizer
        TVizir2 opt_moead(&tds_moead, &runner, &solv_moead);

        // add the objective
        opt_moead.addObjective(&sect); // minimizing the section
        opt_moead.addObjective(&dist); // minimizing the distortion

        opt_moead.addConstraint(&thick,&positiv);  //on thickness (thick > 0.4)

        opt_moead.solverLoop(); // running the optimization

        // Stop the slave processes
        runner.stopSlave();


        // Start the graphical part
        // Preaparing canvas
        TCanvas *fig1 = new TCanvas("fig1","Pareto Zone",5,64,1270,667);
  TPad *pad1 = new TPad("pad1","",0,0.03,1,1);
  pad1->Draw();
        pad1->Divide(2,1); pad1->cd(1);

        // extracting data to construct graphs
        double viz[total], moead[total+4]; // There is always one more point in moead
        tds_viz.getTuple()->extractData(viz, total, "x:y:sect:dist","","column");
        tds_moead.getTuple()->extractData(moead, total+4, "x:y:sect:dist","","column");

        TGraph *set_viz = new TGraph(nbPoints, &viz[0], &viz[nbPoints]);
        TGraph *front_viz = new TGraph(nbPoints, &viz[2*nbPoints], &viz[3*nbPoints]);
        set_viz->SetMarkerColor(4); set_viz->SetMarkerStyle(20); set_viz->SetMarkerSize ↩
            (0.8);
        front_viz->SetMarkerColor(4); front_viz->SetMarkerStyle(20); front_viz-> ↩
            SetMarkerSize(0.8);

        TGraph *set_moead = new TGraph(nbPoints+1, &moead[0], &moead[nbPoints+1]);
        TGraph *front_moead = new TGraph(nbPoints, &moead[2*(nbPoints+1)], &moead[3*( ↩
            nbPoints+1)]);
        set_moead->SetMarkerColor(2); set_moead->SetMarkerStyle(20); set_moead-> ↩
            SetMarkerSize(0.8);
        front_moead->SetMarkerColor(2); front_moead->SetMarkerStyle(20); front_moead-> ↩
            SetMarkerSize(0.8);

        // Legend
        TLegend *leg = new TLegend(0.25, 0.75, 0.55, 0.89);
        leg->AddEntry(set_viz,"Vizir algo","p");
        leg->AddEntry(set_moead,"MOEAD algo","p");

        // Pareto Set
        TMultiGraph *set_mg = new TMultiGraph();
        set_mg->Add(set_viz); set_mg->Add(set_moead);
        set_mg->Draw("aP");
        set_mg->SetTitle("Pareto Set"); set_mg->GetXaxis()->SetTitle("x"); set_mg->GetYaxis ↩
            ()->SetTitle("y");
        leg->Draw();
        gPad->Update();

        // Pareto Front
        pad1->cd(2);
        TMultiGraph *front_mg = new TMultiGraph();
        front_mg->Add(front_viz); front_mg->Add(front_moead);
        front_mg->Draw("aP");
        front_mg->SetTitle("Pareto front"); front_mg->GetXaxis()->SetTitle("Section");  ↩
            front_mg->GetYaxis()->SetTitle("Distortion");
        leg->Draw();
        gPad->Update();


    }
}
```

The variables are defined as follow:

```
// variables
TAttribute x("x", 0.0, 1.0),
y("y", 0.0, 1.0),
thick("thick"), // thickness
sect("sect"), // section of the pipe
dist("dist"); // distortion
```

where the first two are the input ones while the last ones are computed using the provided code (as explained in Section IX.2.2). This code is configured through these lines

```
// Creating the assessor using the analytical function
TCIntEval code("barAllCost");
code.addInput(&x);
code.addInput(&y);
code.addOutput(&thick);
code.addOutput(&sect);
code.addOutput(&dist);
```

The usual Relauncher construction is followed, using a `TSequentialRun` runner. The first solver is defined in these lines

```
 TVizirGenetic solv_viz;
// Set the size of the population to 150, and a maximum number of evaluation at 15000
solv_viz.setSize(nbPoints,nMax);
```

Combining the runner, solver and dataserver, the master object is created and the objective and constraint are defined. This is done in:

```
// Create the multi-objective constrained optimizer
TVizir2 opt_viz(&tds_viz, &runner, &solv_viz);

// add the objective
opt_viz.addObjective(&sect); // minimizing the section
opt_viz.addObjective(&dist); // minimizing the distortion

// and the constrains
TGreaterFit positiv(0.4);
opt_viz.addConstraint(&thick,&positiv);  //on thickness (thick > 0.4);
```

In a second block a new dataserver is created along with a new genetic solver in these lines:

```
// create the vizir genetic solver
TVizirGenetic solv_moead;
solv_moead.setMoeadDiversity(nbPoints);
solv_moead.setStoppingCriteria(1);
solv_moead.setSize(0, nMax, 200);
```

The idea here is to use the Moead algorithm whose principle in few words is to split the space into a certain numbers of direction intervals (set by the argument in the function `setMoeadDiversity`). This should provide a Pareto front with a better homogeneity in the front member distribution (particularly visible here when the size of the requested ensemble is small). The second method, `setStoppingCriteria(1)` states that the only stopping criteria available is the total number of estimation, allowed in the `setSize` method. Finally, the last function to be called is the `setSize` one, with a peculiar first argument here: the size of the pareto can be chosen but if 0 is put (as done here) the number of elements will be the number of intervals defined previously plus one (the plus one comes from the fact that the elements are created at the edge of every interval, so for 20 intervals, there are 21 edges in total).

The rest of the code is creating the plot shown below in which both Pareto set and front are compared.

XIV.9.4.3 Graph

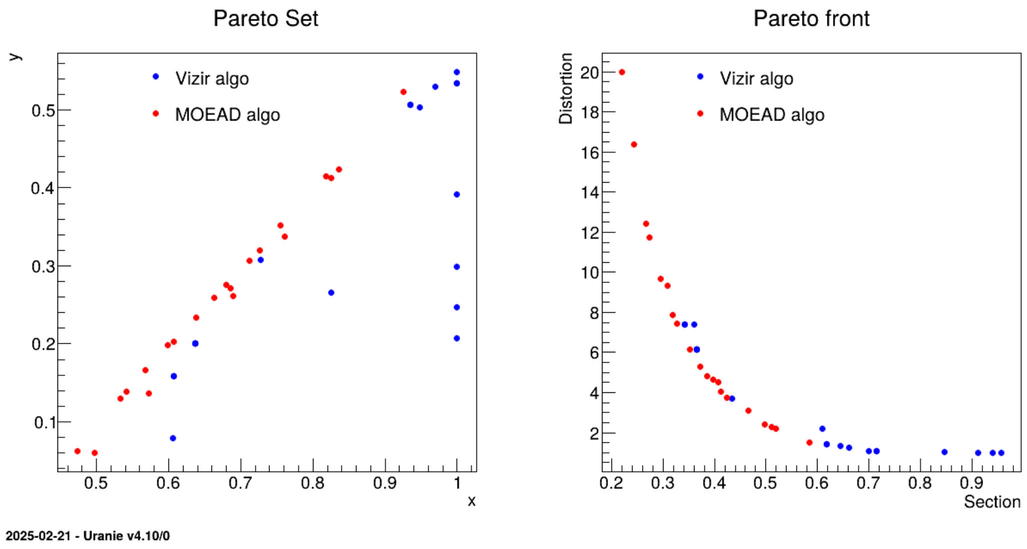


Figure XIV.94: Graph of the macro "reoptimizeHollowBarVizirMoead.C"

XIV.9.5 Macro "reoptimizeHollowBarVizirSplitRuns.C"

XIV.9.5.1 Objective

The objective of the macro is to be able to run an evolutionary algorithm (here we are using a genetic one) with a limited number of code estimation and restart it from where it stopped if it has not converged the first time. This is of utmost usefulness when running a resource-consumming code or (/and) when running on a cluster with a limited number of cpu time. The classical hollow bar example defined in Section IX.2.2 is used to obtain a nice Pareto set/front.

XIV.9.5.2 Macro Uranie

```
#define TOLERANCE 0.001
#define NBmaxEVAL 1200
#define SIZE 500

bool LaunchVizir(int RunNumber, TCanvas *fig1)
{

    // variables
    TAttribute x("x", 0.0, 1.0),
      y("y", 0.0, 1.0),
      thick("thick"),
      sect("sect"),
      def("def");

    TCIntEval code("barAllCost");
    code.addInput(&x);
    code.addInput(&y);
    code.addOutput(&thick);
    code.addOutput(&sect);
    code.addOutput(&def);

    // Create a runner
    TSequentialRun runner(&code);
    runner.startSlave();

    // Output to state whether convergence is reached
    bool hasConverged=false;
    if(runner.onMaster())
    {
        // Create the TDS
        TDataServer tds("vizirDemo", "Param de l'opt vizir pour la barre");
        tds.addAttribute(&x);
        tds.addAttribute(&y);

        TVizirGenetic solv;
        // Name of the file that will contain
        string filename="genetic.dump";
        std::vector<char> cstr(filename.c_str(), filename.c_str() + filename.size() + 1);
        /* Test whether genetic.dump exists. If not, it creates it and returns false, so
        that the "else" part is done to start the initialisation of the vizir algorithm. */
        if ( solv.setResume(NBmaxEVAL, &cstr[0]))
            cout << "Restarting Vizir" << endl;
        else solv.setSize(SIZE, NBmaxEVAL);

        // Create the multi-objective constrained optimizer
        TVizir2 opt(&tds, &runner, &solv);
        opt.setTolerance(TOLERANCE);
        // add the objective
        opt.addObjective(&sect);
        opt.addObjective(&def);
        TGreaterFit positiv(0.4);
        opt.addConstraint(&thick,&positiv);

        /* resolution */
        opt.solverLoop();
        hasConverged=opt.isConverged();
        // Stop the slave processes
        runner.stopSlave();

        fig1->cd(RunNumber+1);
        tds.getTuple()->SetMarkerColor(2);
        tds.draw("def:sect");
        stringstream tit; tit << "Run number "<<RunNumber+1;
        if(hasConverged) tit << ": Converged !";
        ((TH1F*)gPad->GetPrimitive("__tdshisto__0"))->SetTitle( tit.str().c_str() );
    }

    return hasConverged;
}

int reoptimizeHollowBarVizirSplitRuns()
{
    using namespace URANIE::DataServer;
    using namespace URANIE::Relauncher;
    using namespace URANIE::Reoptimizer;

    gROOT->LoadMacro("UserFunctions.C");
    // Delete previous file if it exists
    gSystem->Unlink("genetic.dump");

    bool finished=false;
    int i=0;
    TCanvas *fig1 = new TCanvas("fig1","fig1",1200,800);
    fig1->Divide(2,2);
    while ( ! finished )
    {
        finished=LaunchVizir(i, fig1);
        i++;
    }

   return 1;
}
```

The idea is to show how to run this kind of configuration: the function `LaunchVizir` is the usual script one can run to get an optimisation with Vizir on the hollow bar problem. The aim is to create a Pareto set of 500 points (SIZE) but only allowing 1200 estimation (NBmaxEVAL). With this configuration we are sure that a first round of estimation will not converge, so we will have to restart the optimisation from the point we stopped. With this regard, the beginning of this function is trivial and the main point to be discussed arises once the solver is created.

```
TVizirGenetic solv;
// Name of the file that will contain
string filename="genetic.dump";
std::vector<char> cstr(filename.c_str(), filename.c_str() + filename.size() + 1);
/* Test whether genetic.dump exists. If not, it creates it and returns false, so
that the "else" part is done to start the initialisation of the vizir algorithm. */
if ( solv.setResume(NBmaxEVAL, &cstr[0]))
cout << "Restarting Vizir" << endl;
else solv.setSize(SIZE, NBmaxEVAL);
```

Clearly here, the interesting part apart, from the definition of the name of the file in which the final state will be kept, is the first test on the solver, before using the `setSize` method. A new methods called `setResume` is called, with two arguments : the number of elements requested in the Pareto set and the name of the file in which to save the state or to restart from. This method returns "true" if `genetic.dump` is found and "false" if not. In the first case, the code will assume that this file is the result of a previous run and it will start the optimisation from the its content trying to get all the population non-dominated (if it's not yet the case). If, on the other hand, no file is found, then the code knows that it would have to store the results of its process, in a file whose name is the second argument, and because the function returns "false", then we move to the "else" part, that starts the optimisation.

Apart from this, the rest of the function is doing the optimisation, and plotting the pareto front in a provided canvas. The only new part here is the fact that the solver (its master in fact) is now able to tell whether it has converged or not through the following method

```
hasConverged=opt.isConverged();
```

this argument being return as the results of the function.

This macro contains another function called `reoptimizeHollowBarVizirSplitRuns` which plays the role of the user in front of a ROOT-console. It defines the correct namespace, loads the function file and destroys previously existing `genetic.dump` files. From there it runs the `LaunchVizir` function as many times as needed (thanks to the boolean returned) as the used would do, by restarting the macro, even after exiting the ROOT console.

The plot shown below represent the Pareto front every time the genetic algorithm stops (at the fourth run, it finally converges !).

XIV.9.5.3 Graph

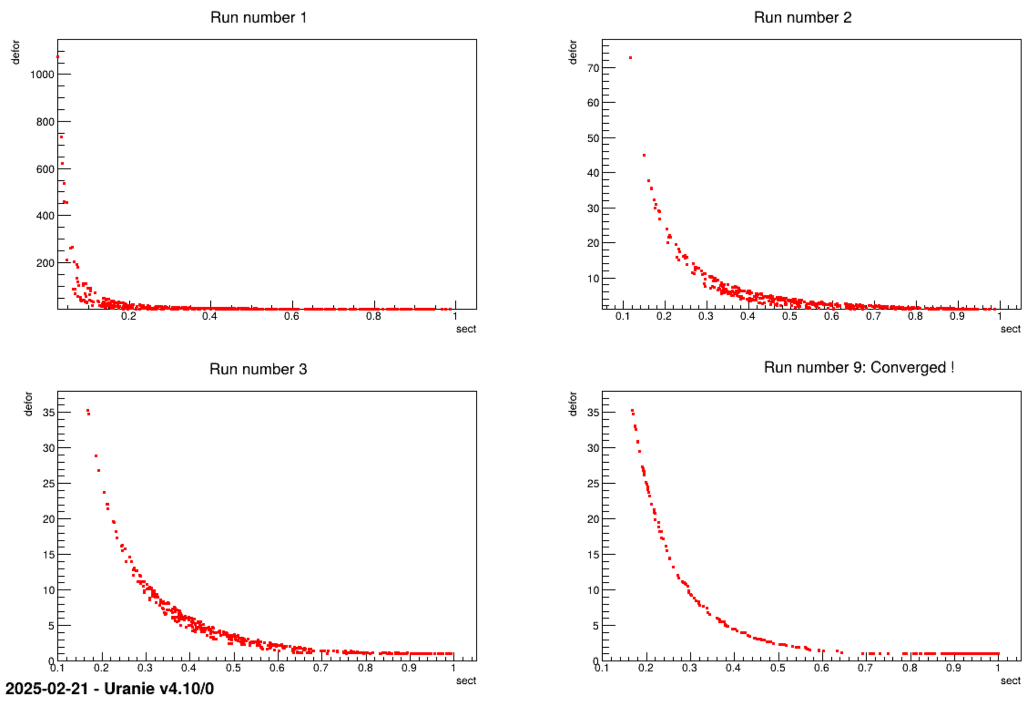


Figure XIV.95: Graph of the macro "reoptimizeHollowBarVizirSplitRuns.C"

XIV.9.6 Macro "`reoptimizeZoningBiSubMpi.C`"

XIV.9.6.1 Objective

The objective of the macro is to show an example of a two level parallelism program using the Mpi paradigm.

- At the top level, an optimization loop parallelizes its evaluations
- At low level, each optimizer evaluation are a launcher loop who parallelizes its own sequential evaluations

These example is inspired from a zoning problem of a small plant core with square assemblies. However, the physics embeded in it is reduced to none (sorry), and the problem is simplified. With symetries, the core is defined by 10 different assemblies presented on the following figure. For production purpose, only 5 assembly types are allowed, defined by an emission value.

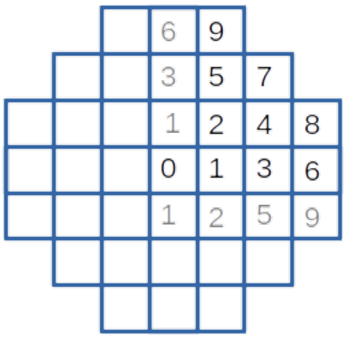


Figure XIV.96: The core and its assemblies

To simplify the problem, some constraints are put :

- most assemblies belong to a default zone
- other zone is restricted to one assembly (or two for 4 and 5, and for 8 and 9 for symetrical reason)
- one zone is imposed with the 8th et 9th external assemblies.
- the total of assembly emission is defined.

For each assembly, a reception value is defined depending on the emission from itself and its neighbour's (just 8 neightbours are taken in account, the 4 nearest neighbours and 4 secondary neighbours). The global objective is to minimize the difference between the biggest and the smallest reception value.

Optimisation works on 4 emission values (the fifth value, affected to the external zone, is set, and all values are normalized with the total emission value) and each evaluation loops over the 35 possible arrangements (choose 3 zones from 7). A single evaluation take emission values and the selected zones and return the maximum reception difference.

XIV.9.6.2 Macro Uranie

This macro is splited in 2 files : the first one defines the low level evaluation function and is reused in the next reoptimizer example. It is quite a mock function, and is given to be complete, but is not needed to understand how to implement the two level MPI parallelism

```
/*
the different zones

6 9
3 5 7
1 2 4 8
0 1 3 6
1 2 5 9

*/

// 4 primary neighbours of a zone
int near1[10][4] = {
  {1,1,1,1}, {0,2,2,3}, {1,1,4,5}, {1,4,5,6}, {2,3,7,8}, // 0-4
  {2,3,7,9}, {3,8,9,10}, {4,5,10,10}, {4,6,10,10}, {5,6,10,10}  // 5-9
};

// 4 secondary neighbours
int near2[10][4] = {
  {2,2,2,2}, {1,1,4,5}, {0,3,3,7}, {2,2,8,9}, {1,5,6,10}, //4
  {1,4,6,10}, {4,5,10,10}, {2,8,9,10}, {3,7,10,10}, {3,7,10,10}  //9
};

// low level evaluation

void lowfun(double *in, double *out)
// evaluate a zoning
{
    double dft;
    double all, min, max, next;
    int p, i, j, id;
    double loc[11], bar[11];

    // init dft
    dft = in[0];
    for (i=0; i<8; i++) loc[i] = dft;
    loc[8] = loc[9] = 0.8;
    loc[10] = 0.;
    // init spec
    for (i=4; i<7; i++) {
        id = (int) in[i];
        loc[id] = in[i-3];
        if (id == 4) loc[5] = in[i-3];
    }
    // normalize
    all = loc[0]/4;
    for (i=1; i<11; i++) all += loc[i];
    for (i=0; i<11; i++) loc[i] *= 10/all;

    // max diff
    i=0;
    next = loc[i];
    for (j=0; j<4; j++) next += loc[near1[i][j]]/8;
    for (j=0; j<4; j++) next += loc[near2[i][j]]/16;
    max = min = next;

    for (i=1; i<10; i++) {
        next = loc[i];
        for (j=0; j<4; j++) next += loc[near1[i][j]]/8;
        for (j=0; j<4; j++) next += loc[near2[i][j]]/16;
        if (next < min) min = next;
        else if (next > max) max = next;
    }

    out[0] = max-min;
    for (i=0; i<10; i++) out[i+1] = loc[i];
}
```

The `lowfun` function deals, as expected, with the low level evaluation. In inputs it has the 4 emission values (default, zone1, zone2, zone3) and 3 indicators defining the zone affected by the extra emission value. It returns the maximal difference between two zone reception values and the 9 normalized emission values (informative data). Two arrays are used to define the neighbourhood

With the second file, the two level MPI parallelism is defined.

```
using namespace URANIE::DataServer;
using namespace URANIE::Relauncher;
using namespace URANIE::Reoptimizer;
using namespace URANIE::MpiRelauncher;

#include "reoptimizeZoneCore.C"

void tds_resume(TDataServer *tds, TAttribute **att, double *res)
{
    TList leaves;
    TLeaf *leaf;
    int i, j, k, siz;
    double obj, cur;
    std::vector<double> tmp;

    siz = tds->getTuple()->GetEntries();

    // init
    for (i=0; att[i]; i++) {
        leaves.Add(tds->GetTuple()->GetLeaf(att[i]->GetName()));
    }
    tmp.resize(i);

    // search min
    tds->GetTuple()->GetEntry(0);
    obj = ((TLeaf *) leaves.At(0))->GetValue(0);
    k = 0;
    for (i=1; i<siz; i++) {
        tds->GetTuple()->GetEntry(i);
        cur = ((TLeaf *) leaves.At(0))->GetValue(0);
        if (cur < obj) {
            obj = cur;
            k = i;
        }
    }

    // get all results
    TIter nextl(&leaves);
    tds->GetTuple()->GetEntry(k);
    for (j=0; (leaf = (TLeaf *) nextl() ); j++) {
        res[j] = leaf->GetValue(0);
    }

}

int doefun(double *in, double *out)
```

```
{
    double z0, z1, z2, z3;
    int i;

    // const
    z0 = in[0];
    z1 = in[1];
    z2 = in[2];
    z3 = in[3];

    // inputs
    TAttribute zon0("zon0", 0., 1.);
    TAttribute zon1("zon1", 0., 1.);
    TAttribute zon2("zon2", 0., 1.);
    TAttribute zon3("zon3", 0., 1.);
    TAttribute a1("a1");
    TAttribute a2("a2");
    TAttribute a3("a3");
    TAttribute *funi[] = { &zon0, &zon1, &zon2, &zon3, &a1, &a2, &a3, NULL};
    //output
    TAttribute diff("diff");
    TAttribute v0("v0");
    TAttribute v1("v1");
    TAttribute v2("v2");
    TAttribute v3("v3");
    TAttribute v4("v4");
    TAttribute v5("v5");
    TAttribute v6("v6");
    TAttribute v7("v7");
    TAttribute v8("v8");
    TAttribute v9("v9");
    TAttribute *funo[] = {
        &diff, &v0, &v1, &v2, &v3, &v4, &v5, &v6, &v7, &v8, &v9, NULL
    };

    // funlow
    TCJitEval lfun(lowfun);
    for (i=0; funi[i]; i++) lfun.addInput(funi[i]);
    for (i=0; funo[i]; i++) lfun.addOutput(funo[i]);

    // runner
    // TSequentialRun run(&lfun);
    TSubMpiRun run(&lfun);
    run.startSlave();
    if (run.onMaster()) {
        TDataServer tds("doe", "tds4doe");
        tds.keepFinalTuple(kFALSE);
        for (i=4; i<7; i++) tds.addAttribute(funi[i]);
        tds.fileDataRead("reoptimizeZoneDoe.dat", kFALSE, kTRUE, "quiet");

        TLauncher2 launch(&tds, &run);
        launch.addConstantValue(&zon0, z0);
        launch.addConstantValue(&zon1, z1);
        launch.addConstantValue(&zon2, z2);
        launch.addConstantValue(&zon3, z3);
        // run doe
        launch.solverLoop();

        //get critere
        tds_resume(&tds, funo, out);

        run.stopSlave();
    }
    return 1;
}

void reoptimizeZoneBiSubMpi()
{
    //ROOT::EnableThreadSafety();
    int i;

    // inputs
    TAttribute z1("zone1", 0., 1.);
    TAttribute z2("zone2", 0., 1.);
    TAttribute z3("zone3", 0., 1.);
    TAttribute z4("zone4", 0., 1.);
    TAttribute *zo[] = { &z1, &z2, &z3, &z4, NULL };

    // outputs
    TAttribute diff("diff");
    TAttribute v0("v0");
    TAttribute v1("v1");
    TAttribute v2("v2");
    TAttribute v3("v3");
    TAttribute v4("v4");
    TAttribute v5("v5");
    TAttribute v6("v6");
    TAttribute v7("v7");
    TAttribute v8("v8");
    TAttribute v9("v9");
    TAttribute *out[] = {
        &diff, &v0, &v1, &v2, &v3, &v4, &v5, &v6, &v7, &v8, &v9, NULL
    };

    // fonction
    TCJitEval fun(doefun);
    for (i=0; zo[i]; i++) fun.addInput(zo[i]);
    for (i=0; out[i]; i++) fun.addOutput(out[i]);
    fun.setMpi();

    // runner
    //TThreadedRun runner(&fun,8);
    //TSequentialRun runner(&fun);
    TBiMpiRun runner(&fun, 3);
    runner.startSlave();
    if (runner.onMaster()) {
        TDataServer tds("tdsvzr", "tds4optim");
        fun.addAllInputs(&tds);

        //
        TVizirGenetic gene;
        gene.setSize(300, 200000, 100);
        //TVizirIsland viz(&tds, &runner, &gene);
        TVizir2 viz(&tds, &runner, &gene);
        //viz.setTolerance(0.00001);
        viz.addObjective(&diff);

        viz.solverLoop();

        runner.stopSlave();
        tds.exportData("__coeur__.dat");
    }
}
```

This script is structured with 3 functions :

- function `tds_resume` is used by the intermediate function. It receives the `TDataServer` filled, loops on its items and returns an synthetic value. In our case, the minimum value of the reception difference, and the 9 normalized emission values
- function `doefun` is the intermediate evaluation function. It runs the design of experiments containing all 35 possible arrangements and extract the best one. It receives the 4 emission values and used them to complete the `TDataServer` using the `addConstantValue` method.
- function `reoptimizeZoningBiSubMpi` is the top level function who solve the zoning problem

`TBiMpiRun` and `TSubMpiRun` are used to allocate cpus between intermediate and low level. `TBiMpiRun` is used in `reoptimizeZoningBiSubMpi` (top) with an integer argument specifying the number of CPUs dedicated to each intermediate level. In our case (3), with 16 resources request to MPI, they are divided in 5 groups of 3 CPUs, and one CPU is left for the top level master (take care that the number of CPUs requested matches group size (16 % 3 == 1)). The top level Master sees 5 resources for his evaluations. `TSubMpiRun` is used in `doefun` function and gives access to the 3 own resources reserved in top level function.

Running the script is done as usual with MPI :

```
mpirun -n 16 root -l -b -q reoptimizeZoningBiSubMpi.C
```

At the begining of `reoptimizeZoningBiSubMpi` function there is a call to `ROOT::EnableThreadSafety`. It is unusefull in this case, but if we parallelize with threads instead of MPI. If you want to use both threads and MPI, it is recommended to use MPI at top level.

XIV.9.7 Macro "reoptimizeZoneBiFunMpi.C"

XIV.9.7.1 Objective

The objective of the macro is to give another example of a two level parallelism program using MPI paradigm. In the former example MPI function call is implicit using Uranie facilities. In this one, explicit calls to MPI functions is done. It's presented to illustrate the case when the user evaluation fonction is an MPI function.

It takes the former example of zoning problem and adapts it. The intermediate level does not use a `TLauncher2` to run all different arrangements, but encodes it. Each Mpi ressources evaluates different possible arrangements keeping its best one, and Mpi reduce these results to the final result.

XIV.9.7.2 Macro Uranie

The low level evaluation function is the same than in previous example and is not shown again.

```
using namespace URANIE::DataServer;
using namespace URANIE::Relauncher;
using namespace URANIE::Reoptimizer;
using namespace URANIE::MpiRelauncher;

#include "reoptimizeZoneCore.C"
#include "reoptimizeZoneDoe.h"

struct mpiret {
    double val;
    int id;
};

int doefun(double*, double*);

void reoptimizeZoneBiFunMpi()
{
    ROOT::EnableThreadSafety();
    int i;

    // inputs
    TAttribute z1("zone1", 0., 1.);
    TAttribute z2("zone2", 0., 1.);
    TAttribute z3("zone3", 0., 1.);
    TAttribute z4("zone4", 0., 1.);
    TAttribute *zo[] = { &z1, &z2, &z3, &z4, NULL };

    // outputs
    TAttribute diff("diff");
    TAttribute v0("v0");
    TAttribute v1("v1");
    TAttribute v2("v2");
    TAttribute v3("v3");
    TAttribute v4("v4");
    TAttribute v5("v5");
    TAttribute v6("v6");
    TAttribute v7("v7");
    TAttribute v8("v8");
    TAttribute v9("v9");
    TAttribute *out[] = {
        &diff, &v0, &v1, &v2, &v3, &v4, &v5, &v6, &v7, &v8, &v9, NULL
    };

    // fonction
    TCJitEval fun(doefun);
    for (i=0; zo[i]; i++) fun.addInput(zo[i]);
    for (i=0; out[i]; i++) fun.addOutput(out[i]);

    // runner
    //TThreadedRun runner(&fun,8);
    //TSequentialRun runner(&fun);
    TBiMpiRun runner(&fun, 3);
    runner.startSlave();
    if (runner.onMaster()) {
        TDataServer tds("tdsvzr", "tds4optim");
        fun.addAllInputs(&tds);

        //
        TVizirGenetic gene;
        gene.setSize(300, 200000, 100);
        //TVizirIsland viz(&tds, &runner, &gene);
        TVizir2 viz(&tds, &runner, &gene);
        //viz.setTolerance(0.00001);
        viz.addObjective(&diff);

        viz.solverLoop();

        runner.stopSlave();
        tds.exportData("__coeurM__.dat");
    }
}
```

```
int doefun(double *in, double *out)
{
    int i, id, size, fid, tag;
    MPI_Comm comm;
    double z[7], one[11], two[11];
    double *cur, *mem, *swp;
    struct mpiret ret, res;

    comm = URANIE::MpiRelauncher::TBiMpiRun::getCalculMpiComm();
    MPI_Comm_rank(comm, &id);
    MPI_Comm_size(comm, &size);

    // const
    z[0] = in[0];
    z[1] = in[1];
    z[2] = in[2];
    z[3] = in[3];

    // local
    mem = one;
    cur = two;
    i = id;
    z[4] = doe[i][0];
    z[5] = doe[i][1];
    z[6] = doe[i][2];
    lowfun(z, mem);
    for (i = id+size; i<DOESIZE; i+=size) {
        z[4] = doe[i][0];
        z[5] = doe[i][1];
        z[6] = doe[i][2];
        lowfun(z, cur);
        if (cur[0] < mem[0]) {
            swp = mem;
            mem = cur;
            cur = swp;
        }
    }

    // global
    /* where is min */
    ret.val = mem[0];
    ret.id = id;
    MPI_Allreduce(&ret, &res, 1, MPI_DOUBLE_INT, MPI_MINLOC, comm);

    /* get min extra datas */
    if (res.id != 0) {
        if (id == res.id) {
            MPI_Send(mem, 11, MPI_DOUBLE, 0, 0, comm);
        }
        else if (id == 0) {
            MPI_Recv(out, 11, MPI_DOUBLE, res.id, 0, comm, MPI_STATUS_IGNORE);
        }
    }
    else {
        for (i=0; i<11; i++) out[i] = mem[i];
    }
    return 1;
}
```

The top level function (`reoptimizeZoneBiFunMpi`) does not change from the previous example and defines a `TBiMpiRun` instances.

The evaluation MPI fonction (`doefun`) is totaly different. It uses the class method `URANIE::MpiRelauncher::TBiMpiRun:` to get the MPI Communicator object (`MPI_Comm`) dedicated to the calcul ressources. With it, different call to MPI functions can be done : `MPI_Comm_rank` and `MPI_Comm_size` to get the context ; `MPI_Allreduce`, `MPI_Send` and `MPI_Recv` to communicate between calculs ressources.

Note that the evaluation function is predeclared and defined after the top level function. It is a trick for cling (the ROOT jit compiler) to know the MPI function : when it compiles, it sees the `TBiMpiRun` before, and loads MPI librairies. With a real MPI user code, which needs a library, cling cannot be used. User needs extra works to make it run (build a standalone program or a ROOT compatible library), but the principle presented before is suitable.

You may run this script the same way as the precedent example, with the same constraint on the ressource number. Also note that this version is really faster than the previous one, avoiding creation and manipulation of a `TDataServer`.

XIV.10 Macros MetaModelOptim

XIV.10.1 Macro "`metamodoptEgoHimmel.C`"

XIV.10.1.1 Objective

The objective of this macro is to optimize a function using the **Efficient Global Optimization** algorithm (EGO). EGO is well suited for problem when solution evaluations are expensive (efficient) and when problem has local minima (global). In our example, user function is artificially slown down and evaluations are parallelized with threads to point out the standard context of its use.

The user function is inspired by an academic function, Himmelblau. It is a multimodal function used to point out the global search that is realized. The problem have 3 parameters and 4 optimal solutions.

XIV.10.1.2 Macro Uranie>

This macro follows the structure of an reoptimizer macro (see previous chapter) and we take a quick look at generic part. Only specific codes are explained in more details.

```
/** user choice **/
#define LENT 123456789 // slow down

#define NK 40 // minimal evaluation number for initialize a kriging model
#define NR 8 // resource number to be used
#define NC 300 // maximum evaluation number
#define NP 3 // parameter number


/* user function  */
int userfun(double* in, double *out)
{
    double him, ret, tmp;
    int i;

    // himmelblau
    ret = 0;
#ifdef LENT
    for (int j=0; j<LENT; j++)
#endif
```

```
    {
        him = 0;
        tmp = in[0]*in[0] - in[1] - 6;
        him += tmp*tmp;
        for (i=1; i<NP; i++) {
            tmp = (i%2) ? in[i]*in[i] - in[i-1] - 6 : in[i] - in[i-1];
            him += tmp*tmp;
        }
        if (him > ret) ret = him;
    }

    out[0] = ret;
    return 1;
}

void metamodoptEgoHimmel()
{
    int i;

    // input and aoutput variables
    URANIE::DataServer::TUniformDistribution x1("x1", -8., 8.);
    URANIE::DataServer::TUniformDistribution x2("x2", -8., 8.);
    URANIE::DataServer::TUniformDistribution x3("x3", -8., 8.);
    URANIE::DataServer::TAttribute *inatt[4] = {&x1, &x2, &x3, NULL};
    URANIE::DataServer::TAttribute y("y");

    // user fun
    URANIE::Relauncher::TCJitEval fun(&userfun);
    for (i=0; inatt[i] != NULL; i++) fun.addInput(inatt[i]);
    fun.addOutput(&y);

    // runner
    URANIE::Relauncher::TThreadedRun run(&fun, NR+1);
    run.startSlave();
    if (run.onMaster()) {
        // tds
        URANIE::DataServer::TDataServer tds("tds", "ego tds");
        tds.keepFinalTuple(kFALSE);
        for (i=0; inatt[i] != NULL; i++) tds.addAttribute(inatt[i]);

        tds.fileDataRead("lhs3.dat", kFALSE, kTRUE);

        // meta modele to use
        URANIE::MetaModelOptim::TEgoKBModeler egomod;
        egomod.setModel("matern7/2", "const", 1e-8);
        egomod.setSolver("ML", "Bobyqa", 300, 800);

        // ei optimiser
        URANIE::MetaModelOptim::TEgoHjDynSolver hjsolv;
        hjsolv.setSize(128, 32);

        // master
        URANIE::MetaModelOptim::TEGO egosolv(&tds, &run);
        egosolv.setSize(NK, NC);
        egosolv.setModeler(&egomod);
        egosolv.setSolver(&hjsolv);
        egosolv.addObjective(&y);

        // run master
        egosolv.solverLoop();
        // results
        tds.exportData("egoC.dat");
        run.stopSlave();
    }
}
```

At beginning, some cpp macro value are defined.

- The `LENT` value is used to slow down evaluation time.
- `NR` value defines the number of threads that are used for evaluation.
- `NP`, `NK` and `NC` values define respectively the number of problem parameters, the number of evalutions needed before creating a first *surrogate model* and the maximum number of evaluation allowed.

The `userfun` function defines the user function, and `metamodoptEgoHimmmel` the optimisation to realize. The latter follows usual structure: It defines the problem variables, a `TCjitEval` to describes the user function, a `TThreadRun` to use thread parallelism. In the master block, a working `TDataServer` is defined, and optimization classes.

Notice that the input variables are defined with a `TUniformDistribution` (not just a `TAttribute` with minimum and maximum values). It may be unuseful in this case where the `TDataServer` is filled with `fileDataRead` but otherwise, a sampler is run implicitely.

EGO uses two different solvers : one for *surrogate model* construction, one for EI optimisation :

- for *surrogate model* only one class is provided (`TKBModeler`). It allows to configure which kriging model to use (`setModel`), and how it is constructed (`setSolver`)
- for maximizing EI, a `TEgoHjDynSolver` is defined, meaning it uses dynamic optimization with the HJMA algorithm. Two parameters are given, first one configuring the initial search, and the second the following one using previous results.

These solvers are passed to the `TEGO` class with a dedicated method. Currently, EGO runs in verbose modes.

The resulting `TDataServer` is filled with all evaluated solutions. In our case of a multi modal problem, keeping just the best solution is not appropriate. A postprocessing is needed to get best solutions.

XIV.11 Macros Calibration

This section introduces few examples dealing with calibration in order to illustrate the different techniques introduced in Chapter XI and few of the options available as well.

XIV.11.1 Macro "`calibrationMinimisationFlowrate1D.C`"

XIV.11.1.1 Objective

The purpose here is to calibrate the value of H_l that entered the `flowrate` model, when only two inputs have been varied (r_ω and L) while the rest of the variables are set to a frozen value: $r = 25050$, $T_u = 89335$, $T_l = 89.55$, $H_u = 1050$, $K_\omega = 10950$. The context has been already discussed in Section XI.2.4 (including discussing the model, here using the model as a C++-function and the first few lines defining the `TDataServer` objects). This macro shows how to use simple minimisation technique, with a **Relauncher**-architecture.

XIV.11.1.2 Macro Uranie

```
{
  // Load the function flowrateCalib1D
  gROOT->LoadMacro("UserFunctions.C");

  // Input reference file
  TString ExpData="Ex2DoE_n100_sd1.75.dat";

  // define the reference
  TDataServer *tdsRef = new TDataServer("tdsRef","doe_exp_Re_Pr");
  tdsRef->fileDataRead(ExpData.Data());

  // define the parameters
  TDataServer *tdsPar = new TDataServer("tdsPar","pouet");
  tdsPar->addAttribute( new TAttribute("hl", 700.0, 760.0) );
  tdsPar->getAttribute("hl")->setDefaultValue(728.0);

  // Create the output attribute
  TAttribute *out = new TAttribute("out");

  // Create interface to assessors
  TCIntEval *Model = new TCIntEval("flowrateCalib1D");
  Model->addInput(tdsPar->getAttribute("hl"));
  Model->addInput(tdsRef->getAttribute("rw"));
  Model->addInput(tdsRef->getAttribute("l"));
  Model->addOutput(out);

  // Set the runner
  TSequentialRun *runner = new TSequentialRun(Model);

  // Set the calibration object
  TMinimisation *cal = new TMinimisation(tdsPar,runner,1);
  cal->setDistanceAndReference("LS",tdsRef,"rw:l","Qexp");
  TNloptSubplexe solv;
  cal->setOptimProperties(&solv);
  cal->estimateParameters();

  // Draw the residuals
  TCanvas *canRes = new TCanvas("CanRes","CanRes",1200,800);
  TPad *apad = new TPad("apad","apad",0, 0.03, 1, 1);  apad->Draw();  apad->cd();
  cal->drawResidues("Residual title","*","","nonewcanvas");

}
```

Apart from the first lines discussed in Section XI.2.4, the important line is the one defining the starting point of the minimisation. This can be done by calling the `setStartingPoint` method of the `TNlopt` class, or simply by defining default value for the parameter attributes. This is done here:

```
tdsPar->getAttribute("hl")->setDefaultValue(728.0);
```

This macro continues by defining the model and the way to run it. The instance created here, is a `TCIntEval` which simply request the three input variables discussed above **in the correct order**. Here the first one has to be H_l, the parameter that we want to calibrate, because of the way the C++-function has been defined and then the two varying ones, (r_ω and L) whose values are coming from the reference input file. Once done, the output attribute is added (as our model computes only one variable) and the chosen distribution strategy is chosen to be sequential.

```
// Create interface to assessors
TCIntEval *Model = new TCIntEval("flowrateCalib1D");
Model->addInput(tdsPar->getAttribute("hl"));
```

```
Model->addInput(tdsRef->getAttribute("rw"));
Model->addInput(tdsRef->getAttribute("l"));
Model->addOutput(out);

// Set the runner
TSequentialRun *runner = new TSequentialRun(Model);
```

Once done the calibration object (`TMinimisation`) is created and, as discussed in Section XI.2.2.1, the first object to be created is the distance function (here the least-square one) through the `setDistanceAndReference`, that also defines the `TDataServer` that contains reference data, the name of the reference inputs and the reference variable to which the output of the model should be compared with. Finally the optimisation algorithm is defined by creating an instance of `TNloptSubplexe` and then the parameters are estimated.

```
// Set the calibration object
TMinimisation *cal = new TMinimisation(tdsPar,runner,1);
cal->setDistanceAndReference("LS",tdsRef,"rw:l","Qexp");
TNloptSubplexe solv;
cal->setOptimProperties(&solv);
cal->estimateParameters();
```

The final part is how to represents part of the results. As this method is a point-estimation there is only one value so it is always displayed on screen, as shown in Section XIV.11.1.3. The other interesting point is to look at the residual, as discussed in [30] and this is done in Figure XIV.97 which shows normally-distributed residual for the *a posteriori* estimations.

XIV.11.1.3 Console

```
Processing calibrationMinimisationFlowrate1D.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

|....:....|....:....|....:....|....:
.************************************************
*    Row   * tdsPar__n *     hl.hl * agreement *
************************************************
*        0 *         0 * 749.72363 * 18.149368 *
************************************************
```

XIV.11.1.4 Graph

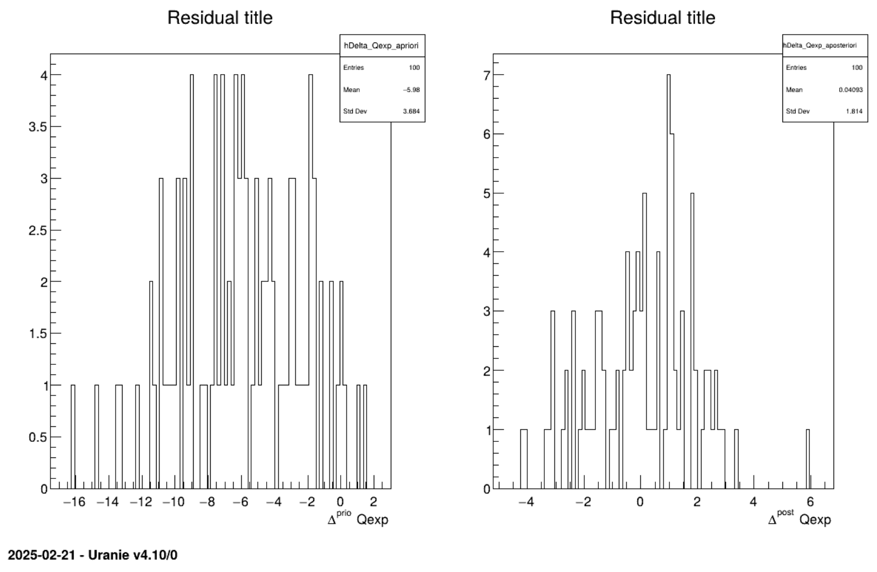


Figure XIV.97: Graph of the macro "`calibrationMinimisationFlowrate1D.C`"

XIV.11.2 Macro "`calibrationLinBayesFlowrate1D.C`"

XIV.11.2.1 Objective

The purpose here is to calibrate the value of H_l that entered the `flowrate` model, when only two inputs have been varied (r_ω and L) while the rest of the variables are set to a frozen value: $r = 25050$, $T_u = 89335$, $T_l = 89.55$, $H_u = 1050$, $K_\omega = 10950$. The context has been already discussed in Section XI.2.4 (including discussing the model, here using the model as a C++-function and the first few lines defining the `TDataServer` objects). This macro shows how to use linear Bayesian estimation technique, with a **Relauncher**-architecture.

XIV.11.2.2 Macro Uranie

```
{
  // Load the function flowrateCalib1D
  gROOT->LoadMacro("UserFunctions.C");

  // Input reference file
  TString ExpData="Ex2DoE_n100_sd1.75.dat";

  // define the reference
  TDataServer *tdsRef = new TDataServer("tdsRef","doe_exp_Re_Pr");
  tdsRef->fileDataRead(ExpData.Data());

  // define the parameters
  TDataServer *tdsPar = new TDataServer("tdsPar","pouet");
  tdsPar->addAttribute( new TUniformDistribution("hl", 700.0, 760.0) );

  // Create the output attribute
  TAttribute *out = new TAttribute("out");

  // Create interface to assessors
  TCIntEval *Reg = new TCIntEval("flowrateModelnoH");
  Reg->addInput(tdsRef->getAttribute("rw"));
  Reg->addInput(tdsRef->getAttribute("l"));
  Reg->addOutput(new TAttribute("H") );
  TSequentialRun *runnoH = new TSequentialRun(Reg);
  runnoH->startSlave();
  if(runnoH->onMaster())
  {
    TLauncher2 l(tdsRef, runnoH);
    l.solverLoop();
    runnoH->stopSlave();
  }

  // Create interface to assessors
  TCIntEval *Model = new TCIntEval("flowrateCalib1D");
  Model->addInput(tdsPar->getAttribute("hl"));
  Model->addInput(tdsRef->getAttribute("rw"));
  Model->addInput(tdsRef->getAttribute("l"));
  Model->addOutput(out);

  // Set the runner
  TSequentialRun *runner = new TSequentialRun(Model);

  // Set the covariance matrix of the input reference
  double sd=tdsRef->getValue("sd_eps",0);
  TMatrixD mat(100,100);
  for(unsigned int ival=0; ival<tdsRef->getNPatterns(); ival++)
    mat(ival,ival)=(sd*sd);

  // Set the calibration object
  TLinearBayesian *cal = new TLinearBayesian(tdsPar,runner,1,"");
  cal->setDistanceAndReference("Mahalanobis",tdsRef,"rw:l","Qexp");
  cal->setObservationCovarianceMatrix(mat);
  cal->setRegressorName("H");
  cal->setParameterTransformationFunction(transf);
  cal->estimateParameters();

  // Draw the residuals
  TCanvas *canRes = new TCanvas("CanRes","CanRes",1200,800);
  TPad *padRes = new TPad("padRes","padRes",0, 0.03, 1, 1);  padRes->Draw();  padRes->cd();
  cal->drawResidues("Residual title","*","","nonewcanvas");

  // Draw the parameters
  TCanvas *canPar = new TCanvas("CanPar","CanPar",1200,800);
  TPad *padPar = new TPad("padPar","padPar",0, 0.03, 1, 1);  padPar->Draw();  padPar->cd();
  cal->drawParameters("Parameter title","*","","nonewcanvas,transformed");
}
```

A very large fraction of this code has been discussed in Section XIV.11.1.2 (from the very start to the sequential run used). The main difference is the fact that the input parameter is now defined as a `TStochasticDistribution`-inheriting object which is the *a priori* chosen distribution. It can, in this case, only be a either a `TNormalDistribution` or a `TUniformDistribution` (see [30]) and the latter is the chosen solution here:

```
tdsPar->addAttribute( new TUniformDistribution("hl", 700.0, 760.0) );
```

Another difference with previous example is the fact that we need to prepare a bit the method as we need to get values for the regressor. As seen in [30], the linear Bayesian estimation is available when the model can be considered linear.

This means that one should linearise the `flowrate` function as done here by writing:

$$f_\theta(x) = (2\pi T_u)\left(\ln(\frac{r}{r_\omega})\left[1+\frac{2LT_u}{\ln(\frac{r}{r_\omega})r_\omega^2 K_\omega}+\frac{T_u}{T_l}\right]\right)^{-1}\theta = H\times\theta$$

where the regressor can be expressed as $H = (2\pi T_u)\left(\ln(\frac{r}{r_\omega})\left[1+\frac{2LT_u}{\ln(\frac{r}{r_\omega})r_\omega^2 K_\omega}+\frac{T_u}{T_l}\right]\right)^{-1}$. From there, it is clear that we will be calibrating a newly defined parameter $\theta = (H_u - H_l)$, so we will have to transform that back into our parameter of interest at some point. To get the regressor estimation we simply use another C++-function `flowrateModelnoH` along with the standard **Relauncher** approach:

```
// Create interface to assessors
TCIntEval *Reg = new TCIntEval("flowrateModelnoH");
Reg->addInput(tdsRef->getAttribute("rw"));
Reg->addInput(tdsRef->getAttribute("l"));
Reg->addOutput(new TAttribute("H") );
TSequentialRun *runnoH = new TSequentialRun(Reg);
runnoH->startSlave();
if(runnoH->onMaster())
{
  TLauncher2 l(tdsRef, runnoH);
  l.solverLoop();
  runnoH->stopSlave();
}
```

Moving on, this method also needs the input covariance matrix. The input datasets provided (`Ex2DoE_n100_sd1.75.dat`) does contain an estimation of the uncertainty affecting the reference measurement, this uncertainty being constant throughout the sample, the input covariance is stated to be diagonal with a constant value set to the standard deviation squared, as, done below:

```
// Set the covariance matrix of the input reference
double sd=tdsRef->getValue("sd_eps",0);
TMatrixD mat(100,100);
for(unsigned int ival=0; ival<tdsRef->getNPatterns(); ival++)
mat(ival,ival)=(sd*sd);
```

The model is anyway defined along with the way to distribute the computation, and then the calibration object is constructed with a Mahalanobis distance function (which is not so relevant as discussed in Section XI.4, as it is only used for illustration purpose). The three important steps are then providing

- the input covariance matrix through the `setObservationCovarianceMatrix` method;
- the regressors name, by calling `setRegressorName`;
- the parameter transformation function (not compulsory) with the `setParameterTransformationFunction`.

The last step is tricky: as we will be performing calibration to get θ, it would be nice to get the proper parameter value in the end. This is possible if one provides a C++-function that transform the parameter estimated from the linearisation back into our parameter of interest. This is done in `UserFunctions.C` for illustration purpose as it contains this `transf` function shown below

```
void transf(double *x, double *res) { res[0] = 1050 - x[0]; } // simply H_l = \theta - H_u
```

The full block of code discussed here is this one

```
// Set the calibration object
TLinearBayesian *cal = new TLinearBayesian(tdsPar,runner,1,"");
cal->setDistanceAndReference("Mahalanobis",tdsRef,"rw:l","Qexp");
cal->setObservationCovarianceMatrix(mat);
cal->setRegressorName("H");
cal->setParameterTransformationFunction(transf);
cal->estimateParameters();
```

The final part is how to represents part of the results. As this method gives normal *a posteriori* laws, there defined by a vector of means and a covariance structure which could simply be accessed and the former is shown in on screen, as shown in Section XIV.11.2.3. Two other *a posteriori* information can be seen as plots: the parameter distribution (shown in Figure XIV.99) and the residual, as discussed in [30], shown in Figure XIV.98 which shows normally-distributed behaviour.

XIV.11.2.3 Console

```
Processing calibrationLinBayesFlowrate1D.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

***  TLinearBayesian::computeParameters(); Transformed parameters estimated are

1x1 matrix is as follows

     |      0    |
------------------
   0 |      749.7
```

XIV.11.2.4 Graph

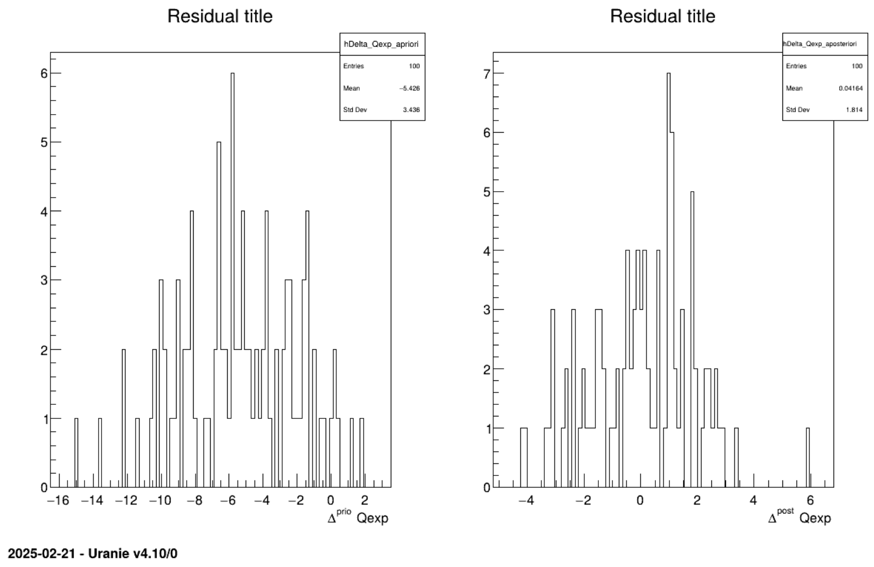


Figure XIV.98: Residual graph of the macro "calibrationLinBayesFlowrate1D.C"

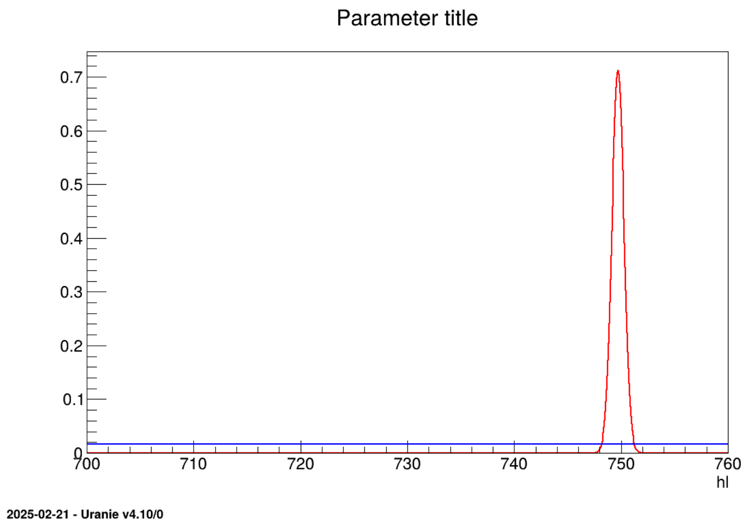


Figure XIV.99: Parameter graph of the macro "calibrationLinBayesFlowrate1D.C"

XIV.11.3 Macro "calibrationRejectionABCFlowrate1D.C"

XIV.11.3.1 Objective

The purpose here is to calibrate the value of H_l that entered the flowrate model, when only two inputs have been varied (r_ω and L) while the rest of the variables are set to a frozen value: $r = 25050$, $T_u = 89335$, $T_l = 89.55$, $H_u = 1050$, $K_\omega = 10950$. The context has been already discussed in Section XI.2.4 (including discussing the model, here using the model as a C++-function and the first few lines defining the TDataServer objects). This macro shows how to use the rejection ABC method, with a **Relauncher**-architecture.

XIV.11.3.2 Macro Uranie

```
{
  // Load the function flowrateCalib1D
  gROOT->LoadMacro("UserFunctions.C");

  // Input reference file
  TString ExpData="Ex2DoE_n100_sd1.75.dat";

  // define the reference
  TDataServer *tdsRef = new TDataServer("tdsRef","doe_exp_Re_Pr");
  tdsRef->fileDataRead(ExpData.Data());

  // define the parameters
  TDataServer *tdsPar = new TDataServer("tdsPar","pouet");
  tdsPar->addAttribute( new TUniformDistribution("hl", 700.0, 760.0) );

  // Create the output attribute
  TAttribute *out = new TAttribute("out");

  // Create interface to assessors
  TCIntEval *Model = new TCIntEval("flowrateCalib1D");
  Model->addInput(tdsPar->getAttribute("hl"));
  Model->addInput(tdsRef->getAttribute("rw"));
  Model->addInput(tdsRef->getAttribute("l"));
  Model->addOutput(out);

  // Set the runner
  TSequentialRun *runner = new TSequentialRun(Model);

  // Set the calibration object
  Int_t nABC = 100; Double_t eps = 0.05;
  TRejectionABC *cal = new TRejectionABC(tdsPar, runner, nABC, "");
  cal->setDistanceAndReference("LS",tdsRef,"rw:l","Qexp");
  cal->setGaussianNoise("sd_eps");
  cal->setPercentile(eps);
  cal->estimateParameters();

  // Compute statistics
  tdsPar->computeStatistic();
  cout << "The mean of hl is " << tdsPar->getAttribute("hl")->getMean() << endl;
  cout << "The std of hl is " << tdsPar->getAttribute("hl")->getStd() << endl;

  // Draw the parameters
  TCanvas *canPar = new TCanvas("CanPar","CanPar",1200,800);
  TPad *padPar = new TPad("padPar","padPar",0, 0.03, 1, 1);  padPar->Draw();  padPar->cd();
  cal->drawParameters("Parameter title","*","","nonewcanvas");

  // Draw the residuals
  TCanvas *canRes = new TCanvas("CanRes","CanRes",1200,800);
  TPad *padRes = new TPad("padRes","padRes",0, 0.03, 1, 1);  padRes->Draw();  padRes->cd();
  cal->drawResidues("Residual title","*","","nonewcanvas");
}
```

A very large fraction of this code has been discussed in Section XIV.11.1.2 (from the very start to the sequential run used). The main difference is the fact that the input parameter is now defined as a `TStochasticDistribution` - inheriting object, as a sample will be generated to test locations:

```
tdsPar->addAttribute( new TUniformDistribution("hl", 700.0, 760.0) );
```

Apart from this, the model is defined along with the way to distribute the computation, and then the calibration object is constructed by defining the number of elements in the final sample (`nABC` set to 100) and, here, the percentile of events kept (`eps` set to 5 percent, which means of total number of estimation of 2000 locations). As the code is deterministic, the uncertainty model is inserted through a random gaussian noise whose standard deviation is defined event-by-event thanks to a variable in the observation dataserver. The distance is also define and the estimation is performed.

```
// Set the calibration object
Int_t nABC = 100; Double_t eps = 0.05;
TRejectionABC *cal = new TRejectionABC(tdsPar, runner, nABC, "");
cal->setDistanceAndReference("LS",tdsRef,"rw:1","Qexp");
cal->setGaussianNoise("sd_eps");
cal->setPercentile(eps);
cal->estimateParameters();
```

The final part is how to represents part of the results. As this method gives a sample, the first two lines give basic statistical information, directly on screen, as shown in Section XIV.11.3.3. Two other *a posteriori* information can be seen as plots: the parameter distribution (shown in Figure XIV.101) and the residual, as discussed in [30], shown in Figure XIV.100 which shows normally-distributed behaviour.

XIV.11.3.3 Console

```
Processing calibrationRejectionABCFlowrate1D.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${SOURCEDIR}/meTIER/calibration/souRCE/TDistanceFunction.cxx] Line ↩
    [601]
 <URANIE::INFO> TDistanceFunction::setGaussianRandomNoise: gaussian random noise(s) is  ↩
    added using information in [sd_eps] to modify the output variable(s) [out].
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${SOURCEDIR}/meTIER/calibration/souRCE/TRejectionABC.cxx] Line ↩
    [107]
 <URANIE::INFO> TRejectionABC::computeParameters:: 2000 evaluations will be performed !
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>
A posteriori mean coordinates are : (749.926)
The mean of hl is 749.926
The std of hl is 1.97042
```

XIV.11.3.4 Graph

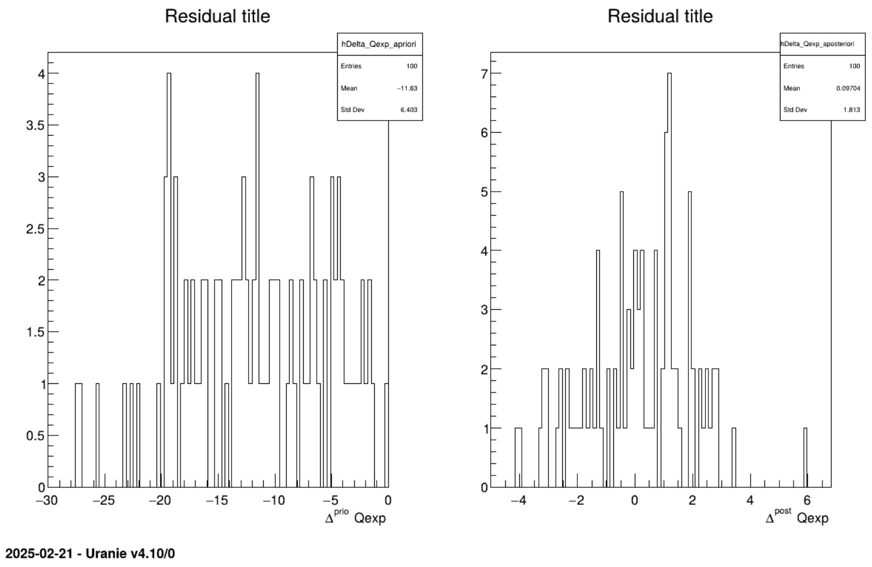


Figure XIV.100: Residual graph of the macro "`calibrationRejectionABCFlowrate1D.C`"

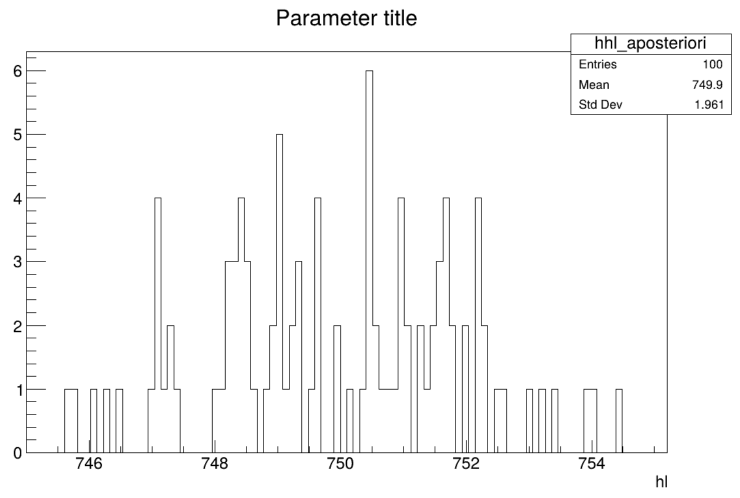


Figure XIV.101: Parameter graph of the macro "`calibrationRejectionABCFlowrate1D.C`"

XIV.11.4 Macro "`calibrationMetropHastingFlowrate1D.C`"

XIV.11.4.1 Objective

The purpose here is to calibrate the value of H_l that entered the `flowrate` model, when only two inputs have been varied (r_ω and L) while the rest of the variables are set to a frozen value: $r = 25050$, $T_u = 89335$, $T_l = 89.55$, $H_u = 1050$, $K_\omega = 10950$. The context has been already discussed in Section XI.2.4 (including discussing the model, here using the model as a C++-function and the first few lines defining the `TDataServer` objects). This macro shows how to use Markov-chain approach with the metropolis-hasting algorithm, using a **Relauncher**-architecture.

XIV.11.4.2 Macro Uranie

```
{
  // Load the function flowrateCalib1D
  gROOT->LoadMacro("UserFunctions.C");

  // Input reference file
  TString ExpData="Ex2DoE_n100_sd1.75.dat";

  // define the reference
  TDataServer *tdsRef = new TDataServer("tdsRef","doe_exp_Re_Pr");
  tdsRef->fileDataRead(ExpData.Data());
  tdsRef->addAttribute("wei_exp","1./(sd_eps*sd_eps)");

  // define the parameters
  TDataServer *tdsPar = new TDataServer("tdsPar","pouet");
  tdsPar->addAttribute( new TUniformDistribution("hl", 700.0, 760.0) );

  // Create the output attribute
  TAttribute *out = new TAttribute("out");

  // Create interface to assessors
  TCIntEval *Model = new TCIntEval("flowrateCalib1D");
  Model->addInput(tdsPar->getAttribute("hl"));
  Model->addInput(tdsRef->getAttribute("rw"));
  Model->addInput(tdsRef->getAttribute("l"));
  Model->addOutput(out);

  // Set the runner
  TSequentialRun *runner = new TSequentialRun(Model);

  // Set the calibration object
  TMetropHasting *cal = new TMetropHasting(tdsPar,runner,2000,"");
  cal->setDistanceAndReference("weightedLS",tdsRef,"rw:l","Qexp","wei_exp");
  cal->setNbDump(400);
  cal->setAcceptationRatioRange(0.4, 0.45);
  cal->estimateParameters();

  // Quality assessment :  Draw the trace the MCMC
  TCanvas *canTr = new TCanvas("CanTr","CanTr",1200,800);
  TPad *padTr = new TPad("padTr","padTr",0, 0.03, 1, 1);  padTr->Draw();  padTr->cd();
  cal->drawTrace("Trace title","*","","nonewcanvas");

  // Draw the parameters
  TCanvas *canPar = new TCanvas("CanPar","CanPar",1200,800);
  TPad *padPar = new TPad("padPar","padPar",0, 0.03, 1, 1);  padPar->Draw();  padPar->cd();
  cal->drawParameters("Parameter title","*","","nonewcanvas");

  // Draw the residuals
  TCanvas *canRes = new TCanvas("CanRes","CanRes",1200,800);
  TPad *padRes = new TPad("padRes","padRes",0, 0.03, 1, 1);  padRes->Draw();  padRes->cd();
  cal->drawResidues("Residual title","*","","nonewcanvas");

  // Compute the auto-correlation
  int burn=20; // Remove first 20 elements
  vector<int> lag={1,5,10,20};
  vector<double> autoCorr;
  cal->getAutoCorrelation(lag, &autoCorr, burn);

  cout<<"Autocorrelation are "<<autoCorr.size()<<":"<<endl;
  for(unsigned il=0; il<lag.size(); il++)
  {
    cout<<"*** for lag="<<lag.at(il)<<": ";
    for(unsigned ip=0; ip<cal->getNPar(); ip++)
    cout<<autoCorr.at(ip*lag.size()+il)<<";  ";
    cout<<endl;
  }
}
```

A very large fraction of this code has been discussed in Section XIV.11.1.2 (from the very start to the sequential run used). The main difference is the fact that the input parameter is now defined as a `TStochasticDistribution` - inheriting object, as a sample will be generated to test locations:

```
tdsPar->addAttribute( new TUniformDistribution("hl", 700.0, 760.0) );
```

Apart from this, the model is defined along with the way to distribute the computation, and then the calibration object is constructed by defining the number of elements in the final sample (set to 2000). The distance function is then defined and two properties are set along: the threshold to which a new line is dump on screen to provide information and the acceptation ratio to be kept by playing on the standard deviation of the research (see Section XI.6). Finally the estimation is performed.

```
// Set the calibration object
TMetropHasting *cal = new TMetropHasting(tdsPar,runner,2000,"");
cal->setDistanceAndReference("weightedLS",tdsRef,"rw:l","Qexp","wei_exp");
cal->setNbDump(400);
cal->setAcceptationRatioRange(0.4, 0.45);
cal->estimateParameters();
```

The final part is how to represents part of the results. At first one should look at the trace to check for any peculiar trend and choose a burn-in threshold if needed (see Section XI.6), which is shown in Figure XIV.102. As this method gives a sample, the first two lines give basic statistical information, directly on screen, as shown in Section XIV.11.4.3. Two other *a posteriori* information can be seen as plots: the parameter distribution (shown in Figure XIV.104) and the residual, as discussed in [30], shown in Figure XIV.103 which shows normally-distributed behaviour.

Finally, the auto-correlation of the resulting sample can be computed with different lag values (see Section XI.6), as by definition, elements from a Markov-chain are not fully independent. This is done here by calling the `getAutoCorrelation` method which provides as many estimations as one request, for the lag values. The results are show on screen (see Section XIV.11.4.3) and are used for post-processing analysis, as the trace plot discussed above.

```
// Compute the auto-correlation
int burn=20;
vector<int> lag={1,5,10,20};
vector<double> autoCorr;
cal->getAutoCorrelation(lag, &autoCorr, burn);

cout<<"Autocorrelation are "<<autoCorr.size()<<":"<<endl;
for(unsigned il=0; il<lag.size(); il++)
{
  cout<<"*** for lag="<<lag.at(il)<<": ";
  for(unsigned ip=0; ip<cal->getNPar(); ip++)
  cout<<autoCorr.at(ip*lag.size()+il)<<"; ";
  cout<<endl;
}
```

XIV.11.4.3 Console

```
Processing calibrationMetropHastingFlowrate1D.C...
```

```
--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

400 events done
800 events done
1200 events done
1600 events done
A posteriori mean coordinates are : (749.66)
Autocorrelation are 4:
*** for lag=1: 0.409977;
*** for lag=5: 0.0365494;
*** for lag=10: 0.0138947;
*** for lag=20: 0.0773913;
```

XIV.11.4.4 Graph

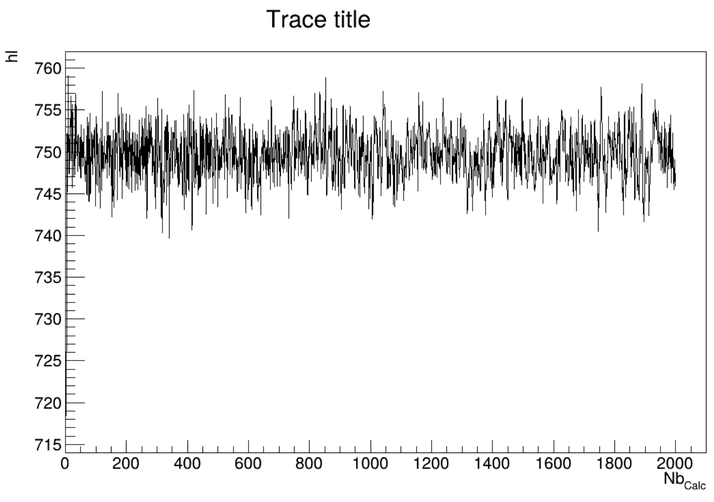


Figure XIV.102: Trace graph of the macro "calibrationMetropHastingFlowrate1D.C"

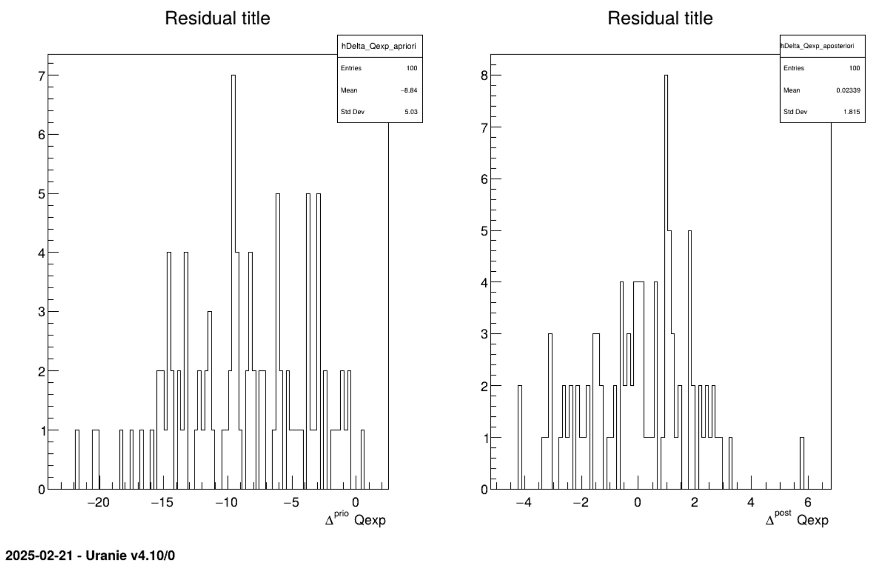


Figure XIV.103: Residual graph of the macro "`calibrationMetropHastingFlowrate1D.C`"

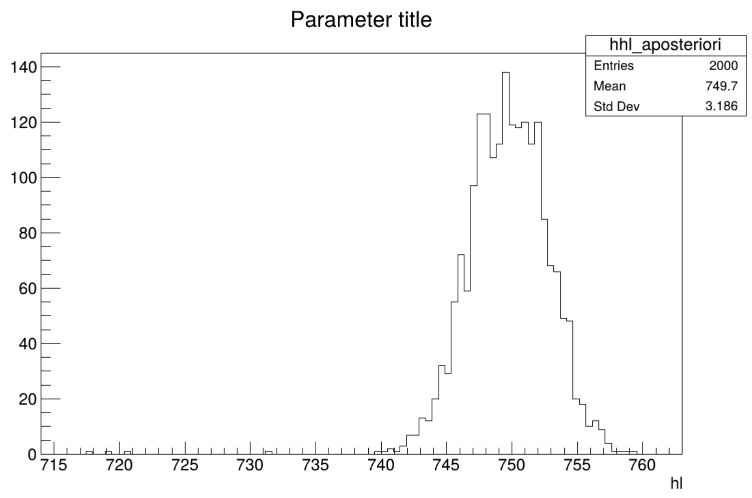


Figure XIV.104: Parameter graph of the macro "`calibrationMetropHastingFlowrate1D.C`"

XIV.11.5 Macro "`calibrationMetropHastingLinReg.C`"

XIV.11.5.1 Objective

The purpose here is to calibrate the value of t_0 and t_1 in a very simple linear model where both parameters are respectively the constant term and the multiplicative coefficient of the only input variable. The input file is `linReg_Database.dat` and the toy model is stored in `UserFunctions.C`. This macro illustrates the steps in two dimensions to get a sample, estimate the burn-in and lag (if needed) and plot the resulting distribution.

XIV.11.5.2 Macro Uranie

```
{
  // Load the function flowrateCalib1D
  gROOT->LoadMacro("UserFunctions.C");

  // Input reference file
  TString ExpData="linReg_Database.dat";

  // Input reference file loaded
  TDataServer *tdsRef = new TDataServer("tdsRef","tdsRef");
  tdsRef->fileDataRead(ExpData.Data());

  // Define the uncertainty model wih a guess
  double sd_exp=0.2;
  tdsRef->addAttribute("wei_exp",Form("1./(%g*%g)",sd_exp,sd_exp));

  // Define the parameters
  TDataServer *tdsPar = new TDataServer("tdsPar","poute");
  double binf_search=-2.0, bsup_search=2.0;
  tdsPar->addAttribute(new TUniformDistribution("t0", binf_search, bsup_search) );
  tdsPar->addAttribute(new TUniformDistribution("t1", binf_search, bsup_search)) ;

  // Create the output attribute
  TAttribute *out = new TAttribute("out");

  // Create interface to assessors
  TCIntEval eval("Toy");
  eval.addInput(tdsRef->getAttribute("x"));
  eval.addInput(tdsPar->getAttribute("t0"));
  eval.addInput(tdsPar->getAttribute("t1"));
  eval.addOutput(out);

  // Set the runner
  TSequentialRun run(&eval);

  // Set the calibration object
  // Providing wild guess for value and variation range
  vector<double> inval={0.8, -0.6}, std={0.4, 0.5};
  int ns=12000;
  TMetropHasting *cal = new TMetropHasting(tdsPar, &run, ns,"");
  cal->setDistanceAndReference("weightedLS",tdsRef,"x","yExp","wei_exp");
  cal->setNbDump(4000);
  cal->setInitialisation(inval, std);
  cal->estimateParameters();

  // Quality assessment :  Draw the trace the MCMC
  TCanvas *canTr = new TCanvas("CanTr","CanTr",1200,800);
  TPad *padTr = new TPad("padTr","padTr",0, 0.03, 1, 1);  padTr->Draw();  padTr->cd();
  cal->drawTrace("Trace title","*","","nonewcanvas");

  // Quality assessment :  Draw the trace the MCMC
  TCanvas *canAcc = new TCanvas("CanAcc","CanAcc",1200,800);
  TPad *padAcc = new TPad("padAcc","padAcc",0, 0.03, 1, 1);  padAcc->Draw();  padAcc->cd();
  cal->drawAcceptationRatio("AcceptRatio title","*","","nonewcanvas");

  int burn=100;
  // Compute the auto-correlation
  vector<int> lag={1,3,6,10,20};
  vector<double> autoCorr;
  cal->getAutoCorrelation(lag, &autoCorr, burn);

  cout<<"Autocorrelation are:"<<endl;
  for(unsigned il=0; il<lag.size(); il++)
  {
    cout<<"*** for lag="<<lag.at(il)<<": ";
    for(unsigned ip=0; ip<cal->getNPar(); ip++)
    cout<<autoCorr.at(ip*lag.size()+il)<<";  ";
    cout<<endl;
  }

  int mylag=6;
  //Define a selection based on burn-in and lag
  TString mycut=Form("(%s > %d) && ((%s %% %d) == 0)", tdsPar->getIteratorName(), burn,   ↩
      tdsPar->getIteratorName(), mylag);
  // Draw the parameters
  TCanvas *canPar = new TCanvas("CanPar","CanPar",1200,800);
  TPad *padPar = new TPad("padPar","padPar",0, 0.03, 1, 1);  padPar->Draw();  padPar->cd();
  cal->drawParameters("Parameter title","*",mycut.Data(),"nonewcanvas");

  // Draw the residuals
  TCanvas *canRes = new TCanvas("CanRes","CanRes",1200,800);
  TPad *padRes = new TPad("padRes","padRes",0, 0.03, 1, 1);  padRes->Draw();  padRes->cd();
  cal->drawResidues("Residual title","*","","nonewcanvas");
}
```

This macro starts, as usual by defining both reference and parameter dataservers. The only specific lines here are these lines used later-on in which we define the uncertainty hypothesis, meaning a guess of the uncertainty by creating the weight variable (constant throughout the 30 reference bservations)

```
// Define the uncertainty model wih a guess
double sd_exp=0.2;
tdsRef->addAttribute("wei_exp",Form("1./(%g*%g)",sd_exp,sd_exp));
```

This macro continues by defining the model and the way to run it. The instance created here, is a `TCIntEval` which simply request the three input variables discussed above **in the correct order**. Here the first one has to be the input variable, whose values are coming from the reference datasets, while the other ones are the parameters to be calibrated, because of the way the C++-function has been defined. Once done, the output attribute is added (as our model computes only one variable) and the chosen distribution strategy is chosen to be sequential.

```
// Create interface to assessors
TCIntEval eval("Toy");
eval.addInput(tdsRef->getAttribute("x"));
eval.addInput(tdsPar->getAttribute("t0"));
eval.addInput(tdsPar->getAttribute("t1"));
eval.addOutput(out);

// Set the runner
TSequentialRun run(&eval);
```

Apart from this, the model is defined along with the way to distribute the computation, and then the calibration object is constructed by defining the number of elements in the final sample (set to 12000). The distance function is then defined and two properties are set along: the threshold to which a new line is dump on screen to provide information and the initialisation properties (values and variation ranges, see Section XI.6). Finally the estimation is performed.

```
// Set the calibration object
// Providing wild guess for value and variation range
vector<double> inval={0.8, -0.6}, std={0.4, 0.5};
int ns=12000;
TMetropHasting *cal = new TMetropHasting(tdsPar, &run, ns,"");
cal->setDistanceAndReference("weightedLS",tdsRef,"x","yExp","wei_exp");
cal->setNbDump(4000);
```

```
cal->setInitialisation(inval, std);
cal->estimateParameters();
```

The final part is how to represents part of the results. At first one should look at the trace to check for any peculiar trend and choose a burn-in threshold if needed (see Section XI.6), which is shown in Figure XIV.105, but one can also look at the acceptation ratio plots show in Figure XIV.106. As this method gives a sample, the first two lines give basic statistical information, directly on screen, as shown in Section XIV.11.5.3. One can also look at the autocorrelation and this might lead to the choice of a lag value to get low autocorrelation values (as shown in the console below).

Given both burn-in and lag values set, two other *a posteriori* information can be seen as plots: the parameter distribution (shown in Figure XIV.108) and the residual, as discussed in [30], shown in Figure XIV.107 which shows normally-distributed behaviour.

XIV.11.5.3 Console

```
Processing calibrationMetropHastingLinReg.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

4000 events done
8000 events done
12000 events done
A posteriori mean coordinates are : (-0.441188,0.373825)
Autocorrelation are:
*** for lag=1: 0.598494;  0.629943;
*** for lag=3: 0.189151;  0.205056;
*** for lag=6: 0.0542879;  0.0667108;
*** for lag=10: 0.0160068;  0.0195018;
*** for lag=20: 0.0363133;  0.00163761;
```

XIV.11.5.4 Graph

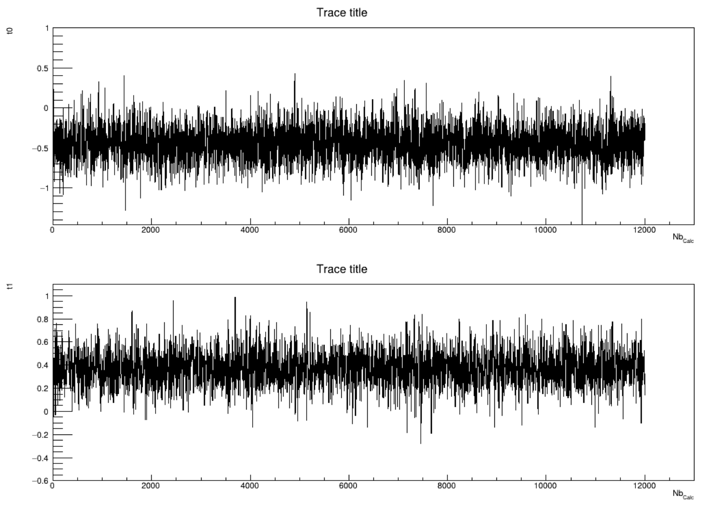


Figure XIV.105: Trace graph of the macro "calibrationMetropHastingLinReg.C"

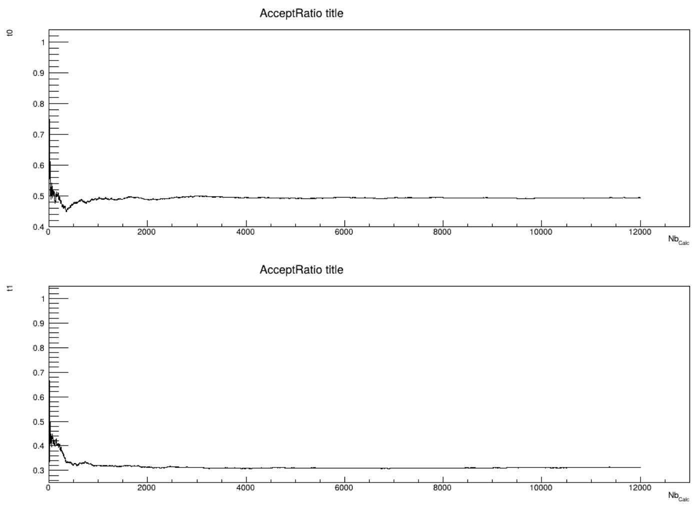


Figure XIV.106: Acceptation rate graph of the macro "`calibrationMetropHastingLinReg.C`"

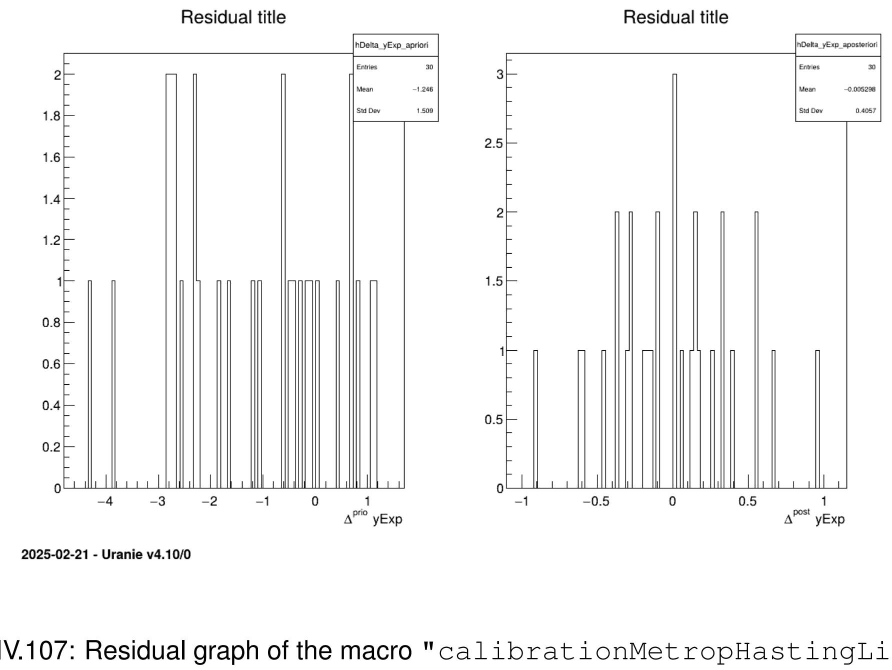


Figure XIV.107: Residual graph of the macro "`calibrationMetropHastingLinReg.C`"

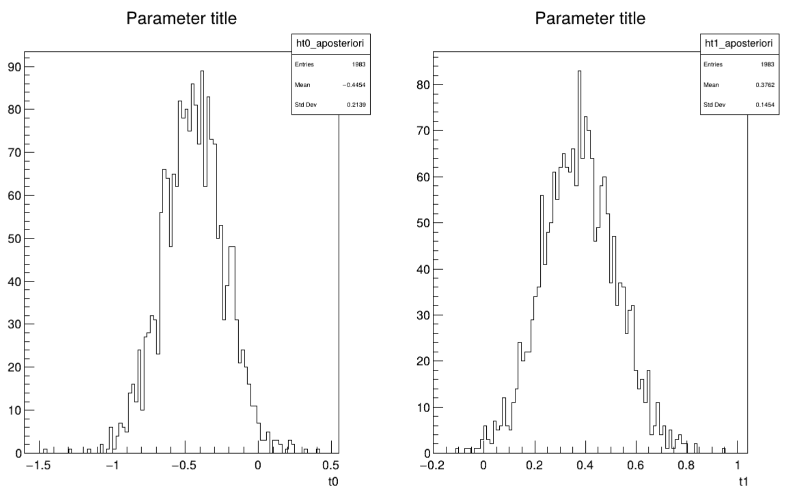


Figure XIV.108: Parameter graph of the macro "calibrationMetropHastingLinReg.C"

XIV.11.6 Macro "calibrationMinimisationFlowrate2DVizir.C"

XIV.11.6.1 Objective

The purpose here is to calibrate the value both of H_u and H_l that entered the `flowrate` model, when only two inputs have been varied (r_ω and L) while the rest of the variables are set to a frozen value: $r = 25050$, $T_u = 89335$, $T_l = 89.55$, $K_\omega = 10950$. The context is the same as the one discussed in Section XIV.11.1 but it describes two things:

- using **Vizir** instead of a more simple `TNloptSolver`-inheriting instance
- discuss the identifiability of a problem, introduced in [30]

The model is the function `flowrateCalib2D` which is the same as the `flowrateClib1D` just requesting the H_u variable as first input.

XIV.11.6.2 Macro Uranie

```
{
  // Load the function flowrateCalib2DVizir
  gROOT->LoadMacro("UserFunctions.C");

  // Input reference file
  TString ExpData="Ex2DoE_n100_sd1.75.dat";

  // define the reference
  TDataServer *tdsRef = new TDataServer("tdsRef","doe_exp_Re_Pr");
  tdsRef->fileDataRead(ExpData.Data());

  // define the parameters
  TDataServer *tdsPar = new TDataServer("tdsPar","pouet");
  tdsPar->addAttribute( new TAttribute("hu", 1020.0, 1080.0) );
  tdsPar->addAttribute( new TAttribute("hl", 720.0, 780.0) );

  // Create the output attribute
  TAttribute *out = new TAttribute("out");

  // Create interface to assessors
  TCIntEval *Model = new TCIntEval("flowrateCalib2D");
  Model->addInput(tdsPar->getAttribute("hu"));
  Model->addInput(tdsPar->getAttribute("hl"));
  Model->addInput(tdsRef->getAttribute("rw"));
  Model->addInput(tdsRef->getAttribute("l"));
  Model->addOutput(out);

  // Set the runner
  TSequentialRun *runner = new TSequentialRun(Model);

  // Set the calibration object
  TMinimisation *cal = new TMinimisation(tdsPar,runner,1);
  cal->setDistanceAndReference("relativeLS",tdsRef,"rw:l","Qexp");
  // Set optimisaiton properties
  URANIE::Reoptimizer::TVizirGenetic solv;
  solv.setSize(24,15000,100);
  cal->setOptimProperties(&solv);
  // ((URANIE::Reoptimizer::TVizir2*)cal->getOptimMaster()->setTolerance(1e-6);
  cal->estimateParameters();

  // Draw the Residual
  TCanvas *canRes = new TCanvas("CanRes","CanRes",1200,800);
  TPad *padRes = new TPad("padRes","padRes",0, 0.03, 1, 1); padRes->Draw(); padRes->cd();
  cal->drawResidues("tutu","*","","nonewcanvas");

  // Draw the box plot of parameters
  TCanvas *canPar = new TCanvas("CanPar","CanPar",1200,800);
  tdsPar->getTuple()->SetMarkerStyle(20); tdsPar->getTuple()->SetMarkerSize(0.8);
  tdsPar->Draw("hu:hl");

  // Look at the correlation and statistic
  tdsPar->computeStatistic("hu:hl");
  TMatrixD corr=tdsPar->computeCorrelationMatrix("hu:hl");
  corr.Print();

  cout<<"hl is "<<tdsPar->getAttribute("hl")->getMean()<<" +- "<<tdsPar->getAttribute("hl") ↩
      ->getStd()<<endl;
  cout<<"hu is "<<tdsPar->getAttribute("hu")->getMean()<<" +- "<<tdsPar->getAttribute("hu") ↩
      ->getStd()<<endl;
}
```

Apart from the first lines discussed in Section XI.2.4, the important line is the one defining the variable, here as `TAttribute` with boundaries to define the phase space in which the algorithm will look for:

```
 tdsPar->addAttribute( new TAttribute("hu", 1020.0, 1080.0) );
tdsPar->addAttribute( new TAttribute("hl", 720.0, 780.0) );
```

This macro continues by defining the model and the way to run it. The instance created here, is a `TCIntEval` which simply request the three input variables discussed above **in the correct order**. Here the first ones have to be H_u and H_l, the parameter that we want to calibrate, because of the way the C++-function has been defined and then the two varying ones, (r_ω and L) whose values are coming from the reference input file. Once done, the output attribute is added (as our model computes only one variable) and the chosen distribution strategy is chosen to be sequential.

```
// Create interface to assessors
TCIntEval *Model = new TCIntEval("flowrateCalib2D");
Model->addInput(tdsPar->getAttribute("hu"));
Model->addInput(tdsPar->getAttribute("hl"));
```

```
Model->addInput(tdsRef->getAttribute("rw"));
Model->addInput(tdsRef->getAttribute("l"));
Model->addOutput(out);

// Set the runner
TSequentialRun *runner = new TSequentialRun(Model);
```

Once done the calibration object (`TMinimisation`) is created and, as discussed in Section XI.2.2.1, the first object to be created is the distance function (here the least-square one) through the `setDistanceAndReference`, that also defines the `TDataServer` that contains reference data, the name of the reference inputs and the reference variable to which the output of the model should be compared with. Finally the optimisation algorithm is defined by creating an instance of `TVizirGenetic` and then the parameters are estimated.

```
// Set the calibration object
TMinimisation *cal = new TMinimisation(tdsPar,runner,1);
cal->setDistanceAndReference("relativeLS",tdsRef,"rw:l","Qexp");
// Set optimisaiton properties
URANIE::Reoptimizer::TVizirGenetic solv;
solv.setSize(24,15000,100);
cal->setOptimProperties(&solv);
// ((URANIE::Reoptimizer::TVizir2*)cal->getOptimMaster()->setTolerance(1e-6);
cal->estimateParameters();
```

The final part is how to represents part of the results. There are many interesting point in this discussion: the residual, which are estimated using the mean of both parameters, are shown in Figure XIV.109. The fact that it is normally-distributed residual for the *a posteriori* estimations shows that the model is correct even though looking at the second plots, the parameters distribution, shows that there is large variety of solutions possible, see Figure XIV.110. This is a problem of identifiability as there are an infinity of solutions that could give the same results, and this can be seen by looking at the correlation matrix shown in Section XIV.11.6.3.

XIV.11.6.3 Console

```
Processing calibrationMinimisationFlowrate2DVizir.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

first 100
Genetic  1

  Generation : 1, rang max 23
  Nb d'evaluation : 100, taille de la Z.P. :  0

  Generation : 2, rang max 23
  Nb d'evaluation : 465, taille de la Z.P. :  1

  Generation : 3, rang max 8
  Nb d'evaluation : 963, taille de la Z.P. :  6

  Generation : 4, rang max 0
  Nb d'evaluation : 1617, taille de la Z.P. :  24
Genetic  converge 1617
************************************************************************************
*    Row   * tdsPar__n *     hu.hu *     hl.hl * agreement * rgpareto. * generatio *
************************************************************************************
*        0 *         0 * 1038.1302 * 738.13383 * 0.3639076 *         0 *         3 *
*        1 *        1 *      1080 *       780 * 0.3639083 *         0 *         3 *
*        2 *        2 * 1040.7058 * 740.77116 * 0.3639018 *         0 *         3 *
*        3 *        3 * 1079.9545 *       780 * 0.3639024 *         0 *         3 *
*        4 *        4 * 1040.7058 * 740.77116 * 0.3639018 *         0 *         1 *
*        5 *        5 * 1038.6638 * 738.70819 * 0.3639025 *         0 *         3 *
*        6 *        6 * 1040.7058 * 740.77116 * 0.3639018 *         0 *         3 *
*        7 *        7 * 1036.9776 * 736.98122 * 0.3639076 *         0 *         3 *
*        8 *        8 * 1036.9776 * 736.98122 * 0.3639076 *         0 *         3 *
*        9 *        9 * 1038.6638 * 738.70819 * 0.3639025 *         0 *         3 *
*       10 *       10 * 1036.9776 * 736.98122 * 0.3639076 *         0 *         2 *
*       11 *       11 *      1080 *       780 * 0.3639083 *         0 *         3 *
*       12 *       12 * 1036.9776 * 736.98122 * 0.3639076 *         0 *         3 *
*       13 *       13 * 1040.7058 * 740.77116 * 0.3639018 *         0 *         2 *
*       14 *       14 *      1080 *       780 * 0.3639083 *         0 *         3 *
*       15 *       15 *      1080 *       780 * 0.3639083 *         0 *         3 *
*       16 *       16 * 1038.6638 * 738.70819 * 0.3639025 *         0 *         3 *
*       17 *       17 * 1040.7058 * 740.77116 * 0.3639018 *         0 *         3 *
*       18 *       18 *      1080 *       780 * 0.3639083 *         0 *         3 *
*       19 *       19 * 1036.9776 * 736.98122 * 0.3639076 *         0 *         3 *
*       20 *       20 * 1036.9042 * 736.96108 * 0.3639019 *         0 *         3 *
*       21 *       21 * 1038.6638 * 738.70819 * 0.3639025 *         0 *         3 *
*       22 *       22 *      1080 *       780 * 0.3639083 *         0 *         3 *
*       23 *       23 *      1080 *       780 * 0.3639083 *         0 *         3 *
************************************************************************************

2x2 matrix is as follows

     |      0    |      1    |
-------------------------------
   0 |          1           1
   1 |          1           1

hl is 752.445 +- 19.9463
hu is 1052.42 +- 19.9598
```

XIV.11.6.4 Graph

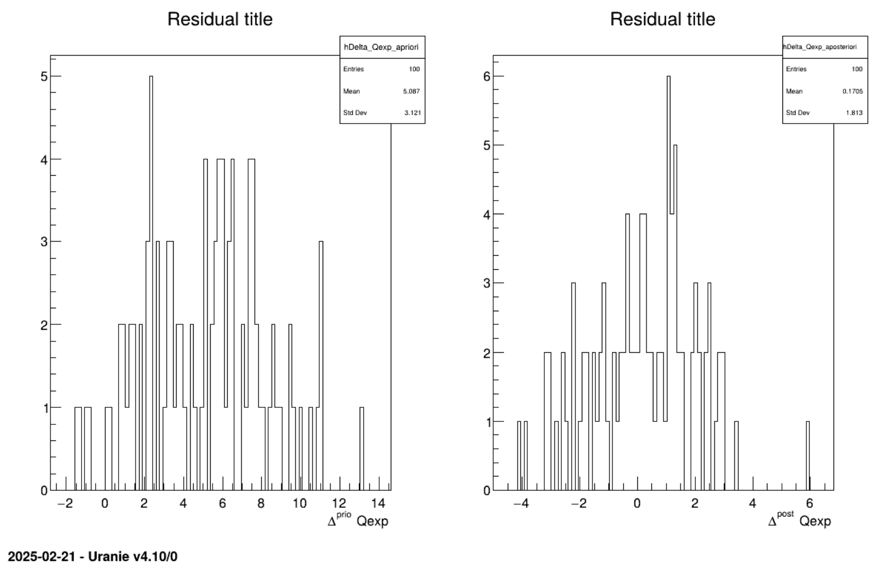


Figure XIV.109: Residual graph of the macro "`calibrationMinimisationFlowrate2DVizir.C`"

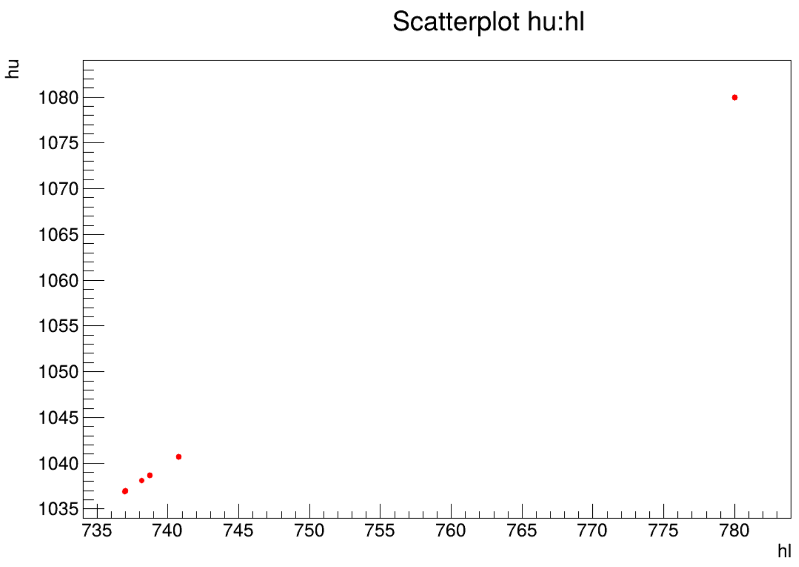


Figure XIV.110: Parameter graph of the macro "`calibrationMinimisationFlowrate2DVizir.C`"

XIV.12 Macros UncertModeler

XIV.12.1 Macro "`uncertModelerTestsYoungsModulus.C`"

XIV.12.1.1 Objective

The objective of the macro is to pass the 3 tests of fit based on Empirical Distribution Function (EDF) statistics (**Kolmogorov-Smirnov (D)**, **Cramer-VonMises (W2)** and **Anderson-Darling (A2)**) on the attribute *"E"* in the "`youngsmodulus`" dataset. The tested law is the **"normal"** distribution when both the mean (30576) and variance (1450) are set or when both are defined either from the sample.

XIV.12.1.2 Macro Uranie

```
{
  TDataServer *tds = new TDataServer();
  tds->fileDataRead("youngsmodulus.dat");

  TCanvas *c = new TCanvas("c1", "Test on youngsmodulus dataset",13,38,1210,1874);
  TPad *pad = new TPad("pad","pad",0, 0.03, 1, 1); pad->Draw();
  pad->Divide(1,3);

  TTestKolmogorovSmirnov *tks = new TTestKolmogorovSmirnov(tds,"E");
  pad->cd(1); tks->computeScore("normal:normal(30576,1450)");

  TTestCramerVonMises *tcvm = new TTestCramerVonMises(tds,"E");
  pad->cd(2); tcvm->computeScore("normal:normal(30576,1450)");

  TTestAndersonDarling *tad = new TTestAndersonDarling(tds,"E");
  pad->cd(3); tad->computeScore("normal:normal(30576,1450)");

}
```

XIV.12.1.3 Graph

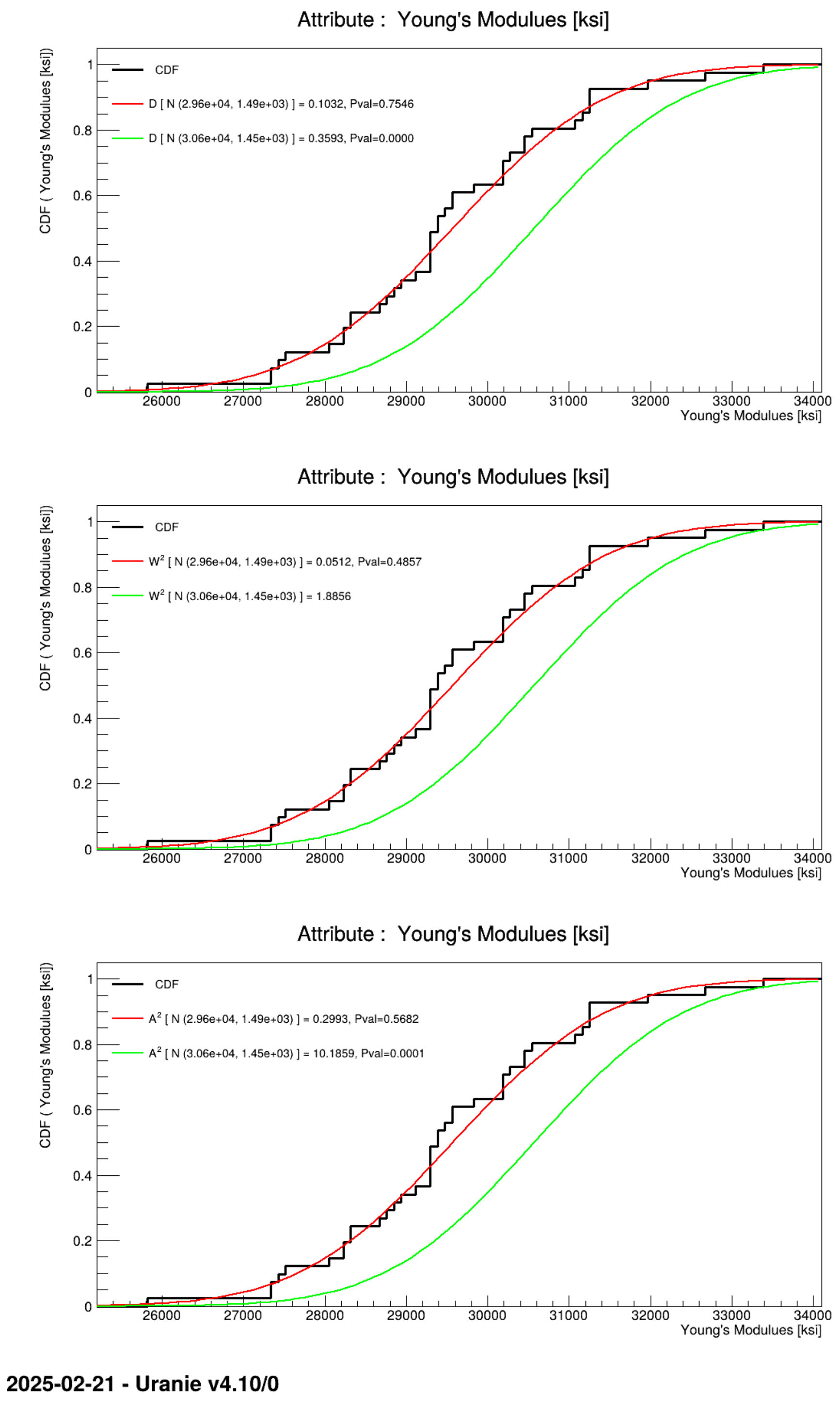


Figure XIV.111: Graph of the macro macro "`uncertModelerTestsYoungsModulus.C`"

XIV.12.2 Macro "`uncertModelerCirce.C`"

XIV.12.2.1 Objective

The objective of the macro **uncertModelerCirce** is to apply the **Circe** method on the dataset "`jdd_circe_summerschool200` `dataserver.dat`", which contains 150 patterns described by 4 attributes (*"code"*,*"exp"* and the derivative from the two parameters *"sens1"* and *"sens2"* of the study).

```
#COLUMN_NAMES: code | exp  | sens1 | sens2

       0.853828       0.720995       1.280695       0.426961
       1.420676       1.467705       2.130798       0.710554
       1.986837       1.277730       2.979664       0.994010
       2.552036       1.991193       3.826800       1.277273
       3.116001       2.036849       4.671714       1.560289
       3.678459       3.445518       5.513915       1.843002
       4.239138       4.735902       6.352916       2.125359
       4.797768       3.381548       7.188232       2.407304
       5.354081       5.383797       8.019378       2.688784
       5.907808       5.001590       8.845874       2.969742
       6.458685       5.330333       9.667245       3.250125
       7.006447       7.952286      10.483015       3.529879
       7.550833       4.561176      11.292717       3.808950
       8.091583       7.968353      12.095884       4.087283
       8.628440       8.644601      12.892057       4.364824
       9.161150       7.772117      13.680780       4.641520
       9.689460      11.946291      14.461602       4.917317
      10.213121       9.110840      15.234080       5.192162
      10.731888       9.179312      15.997775       5.466002
      11.245518      12.044400      16.752254       5.738783
      11.753772       9.955391      17.497091       6.010453
      12.256413       8.516376      18.231867       6.280959
      12.753210      11.832538      18.956171       6.550249
      13.243933      15.764511      19.669597       6.818270
      13.728360      13.636206      20.371749       7.084971
      14.206269      15.828666      21.062237       7.350300
      14.677444      15.371526      21.740682       7.614206
      15.141674      17.624294      22.406711       7.876637
      15.598751      16.365027      23.059960       8.137543
      16.048474      13.622278      23.700075       8.396873
      16.490644      16.654472      24.326711       8.654577
      16.925069      18.445503      24.939533       8.910605
      17.351561      21.030571      25.538214       9.164908
      17.769937      17.357715      26.122438       9.417436
      18.180020      11.956423      26.691900       9.668140
      18.581638      19.157803      27.246304       9.916972
      18.974624      16.126637      27.785365      10.163884
      19.358818      18.922050      28.308810      10.408827
      19.734065      20.848071      28.816374      10.651755
      20.100213      18.048485      29.307807      10.892620
      20.457121       8.858695      29.782866      11.131376
      20.804650      17.677597      30.241324      11.367976
      21.142669      22.920734      30.682962      11.602375
      21.471051      18.370473      31.107575      11.834528
      21.789678      16.656787      31.514968      12.064388
      22.098436      19.602911      31.904959      12.291912
      22.397218      27.669603      32.277379      12.517056
      22.685923      22.298174      32.632070      12.739777
      22.964458      21.384184      32.968887      12.960030
      23.232734      30.519337      33.287695      13.177773
      23.490671      15.001831      33.588376      13.392965
```

```
        23.738192       25.366767       33.870822       13.605563
        23.975231       23.949371       34.134936       13.815527
        24.201726       13.391711       34.380637       14.022815
        24.417621       21.553269       34.607854       14.227387
        24.622868       26.531200       34.816530       14.429205
        24.817425       20.392323       35.006621       14.628229
        25.001258       32.189940       35.178096       14.824419
        25.174337       20.032718       35.330935       15.017740
        25.336642       26.195740       35.465133       15.208152
        25.488157       30.414972       35.580695       15.395619
        25.628873       29.119988       35.677642       15.580104
        25.758789       32.045293       35.756006       15.761573
        25.877910       21.195366       35.815830       15.939990
        25.986247       28.350611       35.857174       16.115319
        26.083817       38.814487       35.880105       16.287529
        26.170646       31.542012       35.884708       16.456584
        26.246764       26.275016       35.871075       16.622452
        26.312208       38.313217       35.839315       16.785102
        26.367023       28.568492       35.789546       16.944501
        26.411259       31.091609       35.721899       17.100619
        26.444972       16.263460       35.636518       17.253425
        26.468224       21.124831       35.533558       17.402891
        26.481085       33.708891       35.413184       17.548986
        26.483629       24.250273       35.275576       17.691683
        26.475938       35.046525       35.120922       17.830954
        26.458098       24.052985       34.949423       17.966773
        26.430201       34.135678       34.761291       18.099112
        26.392347       27.700029       34.556749       18.227946
        26.344640       32.737134       34.336029       18.353251
        26.287189       28.450629       34.099376       18.475001
        26.220109       23.829647       33.847044       18.593174
        26.143522       27.528282       33.579297       18.707747
        26.057552       36.671849       33.296408       18.818696
        25.962331       26.930743       32.998661       18.926002
        25.857996       32.444448       32.686349       19.029643
        25.744687       28.236267       32.359775       19.129598
        25.622549       22.031751       32.019249       19.225849
        25.491734       21.495018       31.665091       19.318378
        25.352397       21.132186       31.297629       19.407165
        25.204696       19.217357       30.917198       19.492194
        25.048797       26.490158       30.524145       19.573449
        24.884866       21.316733       30.118819       19.650913
        24.713076       25.592418       29.701580       19.724572
        24.533603       19.577027       29.272793       19.794412
        24.346625       34.922599       28.832833       19.860418
        24.152327       35.735641       28.382076       19.922578
        23.950895       19.302306       27.920910       19.980881
        23.742519       22.246825       27.449724       20.035314
        23.527392       27.026746       26.968916       20.085868
        23.305709       22.656735       26.478887       20.132532
        23.077671       15.588268       25.980044       20.175297
        22.843477       26.051802       25.472798       20.214156
        22.603332       26.182246       24.957565       20.249100
        22.357443       38.381341       24.434763       20.280123
        22.106018       22.768312       23.904816       20.307219
        21.849267       26.030200       23.368151       20.330382
        21.587402       15.635641       22.825196       20.349609
        21.320639       18.753400       22.276382       20.364895
        21.049191       20.947923       21.722145       20.376237
        20.773276       24.642890       21.162919       20.383634
        20.493113       14.828323       20.599142       20.387083
        20.208919       17.455233       20.031254       20.386585
```

```
      19.920916      16.168686      19.459693      20.382139
      19.629322      15.843812      18.884899      20.373745
      19.334360      18.983964      18.307313      20.361407
      19.036251      20.541221      17.727376      20.345126
      18.735215      14.704089      17.145526      20.324904
      18.431475      27.724483      16.562203      20.300747
      18.125252      20.854847      15.977844      20.272659
      17.816766      15.890789      15.392886      20.240646
      17.506238      16.750030      14.807764      20.204712
      17.193888      10.107342      14.222909      20.164866
      16.879934      15.763192      13.638752      20.121116
      16.564594      27.922521      13.055719      20.073469
      16.248084      14.705787      12.474234      20.021934
      15.930621       9.417969      11.894719      19.966523
      15.612417      13.446139      11.317589      19.907245
      15.293685      13.532392      10.743257      19.844112
      14.974634      20.769703      10.172131      19.777137
      14.655473      17.227635       9.604615      19.706331
      14.336409      12.008431       9.041108      19.631710
      14.017644      17.118440       8.482001      19.553287
      13.699381      14.424335       7.927684      19.471078
      13.381817      14.105580       7.378537      19.385098
      13.065150       8.072573       6.834935      19.295364
      12.749572      10.533065       6.297249      19.201894
      12.435272      11.022880       5.765839      19.104706
      12.122439      12.865744       5.241061      19.003818
      11.811257      10.789248       4.723263      18.899250
      11.501904       5.134292       4.212786      18.791022
      11.194558      11.284947       3.709961      18.679155
      10.889392      10.426379       3.215113      18.563671
      10.586576       8.555081       2.728559      18.444593
      10.286274      10.517881       2.250605      18.321943
       9.988648      12.657006       1.781552      18.195744
       9.693856       9.822359       1.321689      18.066023
       9.402049       9.082523       0.871296      17.932802
       9.113378      12.983850       0.430646      17.796110
       8.827985       6.205202       -2.89e-15     17.655971
```

XIV.12.2.2 Macro Uranie

```
{

  TDataServer *tds = new TDataServer();
  tds->fileDataRead("jdd_circe_summerschool2006_dataserver.dat");

  //  tds->addAttribute("uexp", "0.05*exp");
  //  tds->addAttribute("sens3", "sens1*sens2");

  // Create the TCirce object from the TDS and specify Experimental attribute, Code ↩
      attribute and sensitivity attributes
  TCirce * tc = new TCirce(tds, "exp", "code", "sens1,sens2");
  // tc->setTolerance(1e-5);
  // tc->setYStarSigma("uexp");
  // tc->setNCMatrix(5);

  // TMatrixD initCMat(2,2);
  // initCMat.Zero(); initCMat(0,0) = 0.042737; initCMat(1,1) = 0.525673;
  // tc->setCMatrixInitial(initCMat);

  // TVectorD initBVec(2);
  // initBVec(0) = -1.436394; initBVec(1) = -1.501561;
  // tc->setBVectorInitial(initBVec);
  tc->estimate();

  // Post-treatment
  TVectorD vBiais = tc->getBVector();
  cout << " ************** vBiais rows[" << vBiais.GetNrows() << "]"<< endl;
  vBiais.Print();
  TMatrixD matC = tc->getCMatrix();
  cout << " ************** matC rows[" << matC.GetNrows() << "] col [" << matC.GetNcols()  ↩
      << "]"<< endl;
  matC.Print();
}
```

XIV.12.2.3 Console

```
Processing uncertModelerCirce.C...

--- Uranie v4.10/0 --- Developed with ROOT (6.32.08)
                      Copyright (C) 2013-2025 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Fri Feb 21, 2025

 *********
 ** addData from an another TDS [jdd_circe_summerschool2006_dataserver]
 ** YStar[exp]  YStarSigma[]YHat[code]
 **  Sensitivity Attributes[sens1 sens2]
 ** nparameter [sens1 sens2] size[2]
 ** List Of TDS size[1]
Collection name='TList', class='TList', size=1
 OBJ: URANIE::DataServer::TDataServer jdd_circe_summerschool2006_dataserver _title_
 ** List Of Informations size[3]
Collection name='TList', class='TList', size=3
 OBJ: TNamed  __Circe_YStar_jdd_circe_summerschool2006_dataserver_1__ exp
 OBJ: TNamed  __Circe_YHat_jdd_circe_summerschool2006_dataserver_1__  code
 OBJ: TNamed  __Circe_Sensitivity_jdd_circe_summerschool2006_dataserver_1__ sens1,sens2
 ** End Of addData from an another TDS [jdd_circe_summerschool2006_dataserver]
 *********

 **********************************
 ** Begin Of Initial Matrix C [1/1]

 ** CIRCE HAS CONVERGED
 ** iter[90] ** Likelihood[-2.729559333111159]
 ***** Selected :: iter[0] Likelihood[-2.729559333111159]
 ** matrix C1

2x2 matrix is as follows

     |      0     |      1     |
-------------------------------
   0 |    0.01612           0
   1 |          0     0.03616


 ** vector XM1

Vector (2)  is as follows

     |        1  |
------------------
   0 |-0.0131705
   1 |0.0106155

 ** End Of Initial Matrix C [1/1]
 **********************************
 ** Residual :: Mean [-0.007054035043120376] Std[1.003324966600461]
 ************** vBiais rows[2]

Vector (2)  is as follows

     |        1  |
------------------
   0 |-0.0131705
   1 |0.0106155

 ************** matC rows[2] col [2]

2x2 matrix is as follows

     |      0    |      1    |
-------------------------------
   0 |    0.01612           0
   1 |          0     0.03616
```

XIV.13 Macros Reliability

XIV.13.1 Macro "reliabilityFormSorm.C"

XIV.13.1.1 Objective

The objective of the macro is to perform a FORM SORM study.

This example comes from De Victor's thesis. The problem has three input variables x, fe and M. Each variable follows a normal distribution with the following mean and standard error: x is N(40, 5), fe is N(50, 2.5) and M is N(1000,200). The safety threshold is calculated with M-x*fe and should be positive for a safe solution.

XIV.13.1.2 Macro Uranie

The Script follows the following steps:

- a macro section to choose the optimisation solver;
- a namespace section;
- the definition of the safety function;
- the study procedure.

```
/* possible solver
 * direct
#define SOLV TNloptCobyla
 * lagragien direct
```

```
#define SOLV TNloptBobyqa //doesn't work on this problem
#define SOLV TNloptPraxis
#define SOLV TNloptNelderMead
#define SOLV TNloptSubplexe
 * */
#define SOLV TNloptCobyla

using namespace URANIE::DataServer;
using namespace URANIE::Relauncher;
using namespace URANIE::Reoptimizer;
using namespace URANIE::Reliability;

/********** User Function ****************/
static void fselect(double *in, double *res)
{
  double x,fe,M;

  x=in[0]; fe=in[1]; M=in[2];
  /* critere */
  res[0] = M-x*fe;
}

/********** Optimization  ****************/
void reliabilityFormSorm()
{

  /* inputs */
  TNormalDistribution x("x", 40, 5),
                      fe("x2", 50, 2.5),
                      M("M", 1000, 200);
  /* outputs */
  TAttribute cont("seuil");
  /* starting point */
  vector<double> start{-1., -1., 1.};

  /* code */
  TSimpleTransform fobj;
  fobj.setParameters(3, &x, &fe, &M);

  TCIntEval fcont("fselect");
  fcont.setInputs(3, &x, &fe, &M);
  fcont.setOutputs(1, &cont);

  TFormEval code(&fobj, &fcont);
  TGreaterFit it(0.0);
  code.addConstraint(&cont, &it);

  /* runner */
  TSequentialRun run(&code);
  run.startSlave();
  if (run.onMaster()) {
    /* tds */
    TDataServer tds("toto", "tds for vizir test");
    code.addAllInputs(&tds);

    /*** FORM ***/
    /* optimizer */
    SOLV solv;
    TNlopt nlo(&tds, &run, &solv);
    code.addObjective(&nlo);

    /* resolution */
    nlo.setStartingPoint(start.size(),&start[0]);
    nlo.solverLoop();

    /* results */
    //tds.getTuple()->Scan("*");

    /*** SORM ***/
    TSorm sorm(&tds, &run);
    sorm.solverLoop();

    /* results */
    tds.getTuple()->Scan("*");

    /* cleanup */
    run.stopSlave();
  }
}
```

The study procedure requests the definition of:

- variables: input variables with their statistical laws and the output variable;
- the starting point for the design point optimisation. Take care that it is defined in the normal space (not in the physical space);
- evaluation functions; the transformation function, the safety function, and the composition of both of them;
- a standard sequential `TRun`;
- the `TDataServer` with its inputs declaration;
- the FORM optimisation sequence;
- the SORM estimation sequence;
- back-up and finalisation.

XIV.13.1.3 Console

```
Processing reliabilityFormSorm.C...
|....:....|....:....|....:....|....:....|....:....0050
|....:....|....
*********************************************************************************************************

*    Row   * toto__n__ *   u_x.u_x * u_x2.u_x2 *   u_M.u_M * betaHL.be * form.form *        ↩
   x.x *     x2.x2 *       M.M * seuil.seu * factor.fa * sorm.sorm *
*********************************************************************************************************

*        0 *         0 * -2.289658 * -0.677000 * 1.8963080 * 3.0490734 * 0.0011477 *  ↩
   28.551708 * 48.307498 * 1379.2616 * -1.67e-05 * 1.0203699 * 0.0011711 *
*********************************************************************************************************
```

There are two lines used to show the optimisation progress (evaluation numbers), and then the resulting `TDataServer` is shown. Columns start with two indexes, the three normal variables, the Hasofer-Lind indicator, the FORM estimation, the three physical variables, the FORM correction, and the SORM estimation.

XIV.13.2 Macro "reliabilityFormSormBis.C"

XIV.13.2.1 Objective

This example takes over the previous one, and tries to exploit the machine CPU using threads. For this purpose, FORM will use a gradient optimisation algorithm with a finite differences gradient estimation which may use 2*n+1 CPU, and Sorm is able to use 2*(n-1) CPU. For using thread, the code needs to be compiled *on the fly* using the ROOT facilities.

XIV.13.2.2 Macro Uranie

the main differences between previous Macro are:

- the #include section needed for compilation;
- the use of a gradient optimisation solver;
- the `TCJitEval` (vs the `TCIntEval`) uses to define the safety function;
- the use of `TTreadedRun` for parallelization;
- the use of a `TGradientEstimationRun` object to parallelize finite differences gradient estimation.

```
#include "TAttribute.h"
#include "TNormalDistribution.h"
#include "TLogNormalDistribution.h"
#include "TSimpleTransform.h"
#include "TCJitEval.h"
#include "TFormEval.h"
#include "TThreadedRun.h"
#include "TGradientEstimationRun.h"
#include "TDataServer.h"
#include "TOptimFit.h"
#include "TNloptCobyla.h"
#include "TNloptDirect.h"
#include "TNlopt.h"
#include "TSorm.h"

/* possible solver
 * gradient
#define SOLV TNloptMMA
#define SOLV TNloptSLSQP
 * lagragien gradient
#define SOLV TNloptLBFGS
#define SOLV TNloptNewton
#define SOLV TNloptVariableMetric
 *
 * */
#define SOLV TNloptMMA

using namespace URANIE::DataServer;
using namespace URANIE::Relauncher;
using namespace URANIE::Reoptimizer;
using namespace URANIE::Reliability;

/********** User Function *****************/
static void fselect(double *in, double *res)
{
    double x,fe,M;

    x=in[0]; fe=in[1]; M=in[2];
    /* critere */
    res[0] = M-x*fe;
}

/********** Optimization  *****************/
void reliabilityFormSormBis()
{
    /* inputs */
    TNormalDistribution x("x", 40, 5),
                        fe("x2", 50, 2.5),
                        M("M", 1000, 200);
    /* outputs */
    TAttribute cont("seuil");
    /* starting point */
    double start[] = {-1., -1., 1.};

    /* code */
    TSimpleTransform fobj;
    fobj.setParameters(3, &x, &fe, &M);

    TCJitEval fcont(&fselect);
    fcont.setInputs(3, &x, &fe, &M);
    fcont.setOutputs(1, &cont);

    TFormEval code(&fobj, &fcont);
    TGreaterFit it(0.0);
    code.addConstraint(&cont, &it);

    TThreadedRun trun(&code, 5);
    TGradientEstimationRun run(&trun);
    run.startSlave();
    if (run.onMaster()) {
        /* tds */
        TDataServer tds("toto", "tds for vizir test");
        code.addAllInputs(&tds);

        /*** FORM ***/
        /* optimizer */
        SOLV solv;
        TNlopt nlo(&tds, &run, &solv);
        code.addObjective(&nlo);

        /* resolution */
        nlo.setStartingPoint(start);
        nlo.solverLoop();

        /*** SORM ***/
        TSorm sorm(&tds, &run);

        sorm.solverLoop();

        /* results */
        tds.getTuple()->Scan("*");

        /* cleanup */
        run.stopSlave();
    }
```

```
}
```

XIV.13.2.3 Console

The console is very similar to the previous one.

```
Processing reliabilityFormSormBis.C+...
!,,,,;,,
****************************************************************************************************
*    Row   * toto__n__ *       u_x *      u_x2 *       u_M *    betaHL *      form *  ↩
            x *        x2 *         M *     seuil *    factor *      sorm *
****************************************************************************************************
*        0 *         0 * -2.286037 * -0.677986 * 1.9003293 * 3.0490789 * 0.0011477 *  ↩
    28.569814 * 48.305034 * 1380.0658 * -1.34e-05 * 1.0204395 * 0.0011711 *
****************************************************************************************************
```

You may notice that the evaluation numbers is very low, but each evaluation is equivalent to 7 evaluations. These traces are obtained in a second run: in the first run you will see compiler messages.

Chapter XV

References

[1] N. Gilardi. Interface python pour la plate-forme uranie. Technical report, CEA, SFME/LGLS/RT/09-015/A, 2009.

[2] Damar Wicaksono. Borehole function. `https://uqworld.org/t/borehole-function/60`, 2019.

[3] W. Appel. *Probabilité pour les non probabilistes.* H & K, Paris, 2013.

[4] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 42(1):55–61, February 2000.

[5] J. C. Helton and F. J. Davis. Illustration of sampling-based methods for uncertainty and sensitivity analysis. *Risk Analysis*, 22(3):591–622, 2002.

[6] R. L. Iman and W. J. Conover. A distribution-free approach to inducing rank correlation among input variables. *Communications in Statistics - Simulation and Computation*, 11(3):311–334, 1982.

[7] I.M Sobol'. On the distribution of points in a cube and the approximate evaluation of integrals. *USSR Computational Mathematics and Mathematical Physics*, 7(4):86 – 112, 1967.

[8] J. H. Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Commun. ACM*, 7(12):701–702, December 1964.

[9] K. Petras. Fast calculation of coefficients in the smolyak algorithm. *Numerical Algorithms*, 26(2):93–109, 2001.

[10] M. Baudin and J.M. Martinez. Polynômes de chaos sous Scilab via la librairie NISP. In *42èmes Journées de Statistique*, Marseille, France, France, 2010.

[11] G. Matheron. La théorie des variables régionalisées, et ses applications. *Fasicule 5 in Les Cahiers du Centre de Morphologie Mathématique de Fontainebleau*, 1970.

[12] J.M. Martinez, A. Marrel, N. Gilardi, and F. Bachoc. Krigeage par processus gaussiens. Librairie gpLib. Technical report, CEA DEN/DANS/DM2S/STMF/LGLS/RT/12-026/A, 2012.

[13] F. Bachoc. *Estimation paramétrique de la fonction de covariance dans le modèle de Krigeage par processus Gaussiens : application à la quantification des incertitudes en simulation numérique*. PhD thesis, Mathématiques appliquées, Paris 7, 2013. Thèse de doctorat dirigée par J. Garnier.

[14] T. Homma and A. Saltelli. Importance measures in global sensitivity analysis of nonlinear models. *Reliability Engineering and System Safety*, 52:1–17, 1996.

[15] A. Saltelli. Making best use of model evaluations to compute sensitivity indices. *Computer Physics Communications*, 145:280–297, 2002.

[16] H. Monod, C. Naud, and D. Makowski. *Uncertainty and sensitivity analysis for crop models*. In D. Wallach, D. Makowski, and J. W. Jones, editors, 2006.

[17] Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.

[18] Ian Jolliffe. *Principal component analysis*. Springer, 2011.

[19] J.M. Martinez. Analyse de sensibilité globale par décomposition de la variance. Technical report, GdR Ondes et Mascot Num, institut Henri Poincaré, 2011.

[20] I.M. Sobol'. Sensitivity indices for nonlinear mathematical models. *Mathematical Modelling and Computational Experiment 1*, 1993.

[21] Richard David Wilkinson. Approximate bayesian computation (abc) gives exact results under the assumption of model error. *Statistical applications in genetics and molecular biology*, 12(2):129–141, 2013.

[22] Andrew Gelman, Gareth O Roberts, Walter R Gilks, et al. Efficient metropolis jumping rules. *Bayesian statistics*, 5(599-608):42, 1996.

[23] Gareth O Roberts, Andrew Gelman, Walter R Gilks, et al. Weak convergence and optimal scaling of random walk metropolis algorithms. *The annals of applied probability*, 7(1):110–120, 1997.

[24] William A Link and Mitchell J Eaton. On thinning of chains in mcmc. *Methods in ecology and evolution*, 3(1):112–115, 2012.

[25] M.J.W. Jansen. Analysis of variance designs for model output. *Computer Physics Communications*, 117, 1999.

[26] G.J. McRae, J.W. Tilden, and J.H. Seinfeld. Global sensitivity analysis: a computational implementation of the fourier amplitude sensitivity test (fast). *Computers & Chemical Engineering*, 6(1):15 – 25, 1982.

[27] A. Saltelli and R. Bolado. An alternative way to compute fourier amplitude sensitivity test (fast). *Computational Statistics & Data Analysis*, 26(4):445 – 460, 1998.

[28] S. Tarantola, D. Gatelli, and T.A. Mara. Random balance designs for the estimation of first order global sensitivity indices. *Reliability Engineering & System Safety*, 91(6):717 – 727, 2006.

[29] G. Arnaud. Manuel d'utilisation de Vizir distribué v2.0. Technical report, CEA, SFME/LGLS/RT/10-001/A, 2010.

[30] J-B. Blanchard. Methodological reference guide for uranie v3.11.0. Technical report, CEA, DEN/DANS/DM2S/STMF/LGLS/RT/17-006/A, 2017. *Updated with every new release.*

[31] R. Brun and F. Rademakers. ROOT: An object oriented data analysis framework. *Nucl. Instrum. Meth.*, A389:81–86, 1997.

[32] Ken Martin and Bill Hoffman. An open source approach to developing software in a small organization. *Ieee Software*, 24(1), 2007.

[33] J. C. Meza, R. A. Oliva, P. D. Hough, and P. J. Williams. Opt++: An object-oriented toolkit for nonlinear optimization. *ACM Transactions on Mathematical Software*, 33(2), June 2007.

[34] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[35] Steven G. Johnson. The nlopt nonlinear-optimization package. http://ab-initio.mit.edu/nlopt.

[36] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[37] Michael Feathers and B Lepilleur. Cppunit cookbook, 2002.

[38] CUDA Nvidia. Nvidia cuda c programming guide. *Nvidia Corporation*, 120(18):8, 2011.

[39] Kai-Tai Fang, Runze Li, and Agus Sudjianto. *Design and modeling for computer experiments*. CRC Press, 2005.

[40] Brian A Worley. Deterministic uncertainty analysis. Technical report, Oak Ridge National Lab., TN (USA), 1987.

[41] T Ishigami and Toshimitsu Homma. An importance quantification technique in uncertainty analysis for computer models. In *Uncertainty Modeling and Analysis, 1990. Proceedings., First International Symposium on*, pages 398–403. IEEE, 1990.

[42] HoHo Rosenbrock. An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 3(3):175–184, 1960.