UserManual Python

The Uranie team, CEA DES, support-uranie@cea.fr

Feb 13, 2026

CONTENTS:

LIST OF FIGURES

LIST OF TABLES

Abstract This documentation presents the features of the Uranie platform (based on Uranie *v4.11.0*), that is developed at CEA/DES. This platform is designed for uncertainty propagation, sensitivity analysis and computational code qualification, in a single software environment. It is largely based on the **ROOT** software (http://root.cern.ch/), an oriented-object framework that is designed and maintained by CERN, primarily used by the particle physicist to analyse the very large amount of data recorded at the **LHC** (*Large Hadron Collider*).

Commit
61cebb3

CHAPTER

ONE

OVERVIEW: URANIE IN A NUTSHELL

Abstract This chapter introduces the global organisation of the Uranie software at a very large scale (module one) and its main dependence: ROOT. Each and every Uranie modules is also briefly described in a specific section, before being furthermore discussed later on.

1.1 Introducing Uranie

Uranie (the version under discussion here being v4.11.0) 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 v6.36.06).

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.

1.1.1 Uranie modules organisation

The platform consists of a set of so-called **technical libraries**, or **modules** (represented as green boxes in Figure 1.1), each performing a specific task.

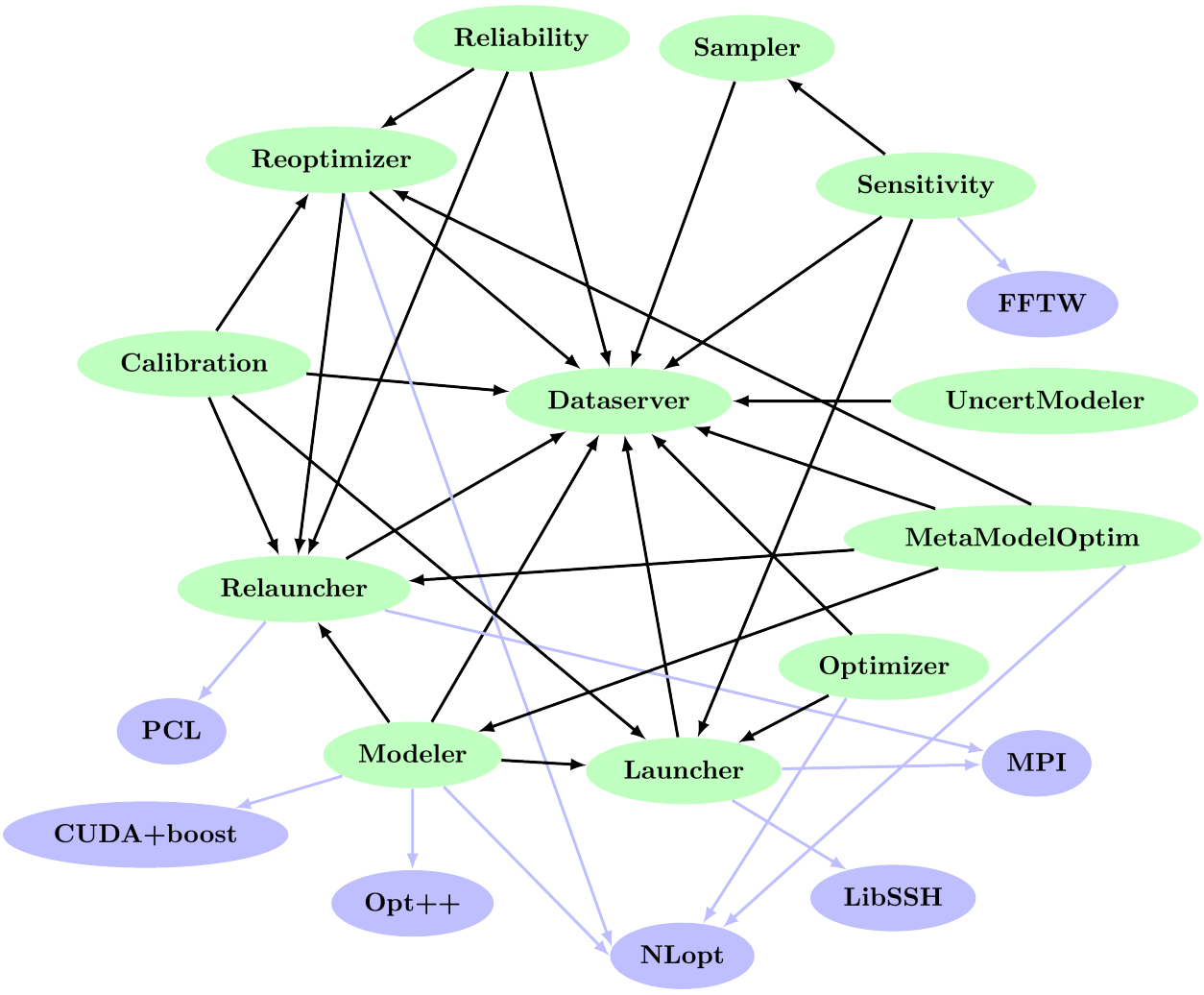


Figure 1.1: Organisation of the Uranie-modules (green boxes) in terms of inter-dependencies. The blue boxes represent the external dependencies (discussed later on).

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 1.1. A more precise description will then be done in the dedicated other chapters.

1.1.1.1 DataServer module

The **DataServer** library (cf *The DataServer module*) 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.

1.1.1.2 Sampler module

The **Sampler** library (cf *The Sampler module*) 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.

1.1.1.3 Launcher module

The **Launcher** library (cf *The Launcher module*) 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*…).

1.1.1.4 Modeler module

The **Modeler** library (cf *The Modeler Module*) 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, …).

1.1.1.5 Sensitivity module

The **Sensitivity** library (cf *The Sensitivity module*) 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 [Bla17].

1.1.1.6 Optimizer and Reoptimizer modules

The **Optimizer** and **Reoptimizer** libraries (cf *The Optimizer module* and *The Reoptimizer module*) 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.

1.1.1.7 Relauncher module

The **Relauncher** library (cf *The Relauncher module*) 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.

1.1.1.8 Calibration module

The **Calibration** library (cf *The Calibration module*) 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.

1.1.2 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 1.1 and both types are listed and briefly discussed below.

1.1.2.1 Compulsory dependencies

ROOT:
: An oriented-object package that offers many possibilities for data handling, analysis, display… [BR97] Sources, binaries and documentation are available at http://root.cern.ch (version used is v6.36.06)

Cmake:
: Free and open-source software for managing the build process of compiled software [MH07]. It is available at http://www.cmake.org/ (version used is v3.28.3)

CPPUnit:
: Unit testing framework for C++ programming [FL02]. It is available at http://sourceforge.net/projects/cppunit/ (version used is v1.15.1)

1.1.2.2 Optional dependencies

OPT++:
Libraries that include non linear optimisation algorithms written in C++ [MOHW07]. 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 [FJ05]. 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 [Joh]. 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 [GFB+04]. 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) [Nvi11] (version used is v13.0). If requested, it should be used with the **boost** library, with a version greater than v1.83.

1.2 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.

1.2.1 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.

1.2.2 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 *Uranie namespace*.

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.

1.2.3 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}`
```

1.2.4 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 *Uranie modules organisation*.

- *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);
```

(continues on next page)

(continued from previous page)

```
    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.

1.2.5 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.

1.2.6 References

The ROOT environment [BR97] 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).

1.3 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 are now available in Python in *Use-cases in Python*.

1.3.1 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 *Use-cases in Python* 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

When typing the following command on many present Linux OS

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

one can get these lines

```
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.

1.3.2 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.

1.3.3 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 *Histogram produced using PyROOT*

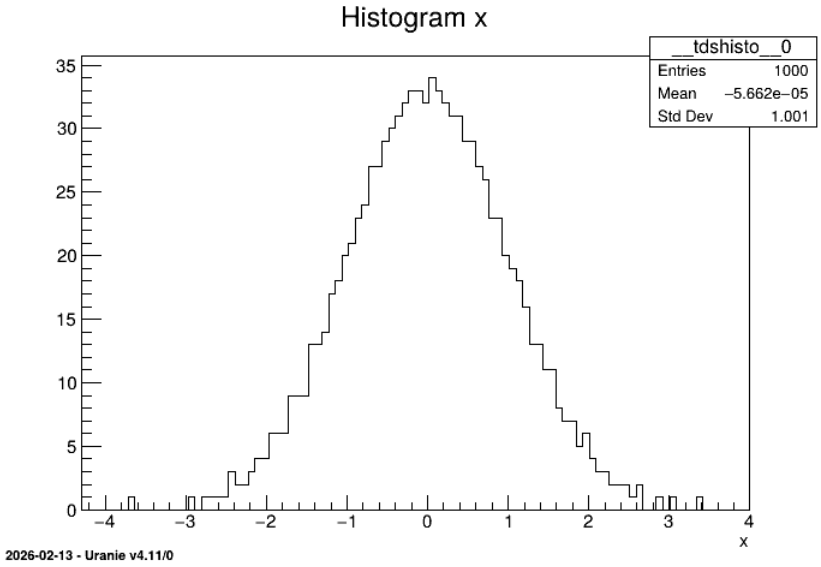


Figure 1.2: Histogram produced using PyROOT

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.

1.3.4 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:

```
# 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
```

(continues on next page)

(continued from previous page)

```
    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()
#
```

1.3.5 FAQ: a Python handbook to interact with Uranie

This section introduces some frequently asked methods to help handling data and objects (as many objects proposed are either ROOT or C++-based). Some of these methods are used throughout the examples in the macros in *Use-cases in Python*, but the aim here is to gather everything that might be of use for a python user that would like to put his hand on Uranie. Most of this methods rely on ROOT version being greater than 6.20 but the modification are small if one uses

older versions (as long as it is greater than6).

1.3.5.1 TDataServer <-> array conversion

This section describes how to convert a `TDataServer` into a `numpy.array` and vice-versa. A breakdown of the macro show below can be found in *Macro "howtoConvertTDataServerArray.py"*

```
"""
Example of TDataServer into numpy array converter
"""
import numpy as np
from URANIE import DataServer

# import data into a dataserver
tds = DataServer.TDataServer("tds", "pouet")
tds.fileDataRead("myData.dat")
print("Dumping tds")
tds.scan()

# Define the list of variable to be read
VarList = "x:y"

# Create an array by defining the shape and the type (this array is a matrix)
TdsData = np.empty(shape=(tds.getNPatterns(), len(VarList.split(":"))),
                   dtype=np.float64)
# Dump the data into the created array (the last field empty unless a select is done␣
↪on the sample)
tds.getTuple().extractData(TdsData, TdsData.size, VarList, tds.getCut().GetTitle())

# Check the content
print("Dumping the array")
print(TdsData)

# Create a new Dataserver and feed it
tds2 = DataServer.TDataServer("brandnew", "pouet")
# Add the attribute in the dataserver and create the tuple
for name in VarList.split(":"):
    index = VarList.split(":").index(name)
    VarData = np.ascontiguousarray(TdsData.transpose()[index])
    tds2.addAttributeUsingData(name, VarData, VarData.size)

print("Dumping new tds")
tds2.Scan("*")
```

The macro shown above, is an example of how to convert the content stored in a `TDataServer` and its result is discussed in *Macro "howtoConvertTDataServerArray.py"*.

1.3.5.2 TMatrixD <-> array conversion

This section describes how to convert a `TMatrixD` into a `numpy.array` and vice-versa. A breakdown of the macro show below can be found in *Macro "howtoConvertTMatrixDArray.py"*

```
"""
Example of TMatrix into numpy array conversion
```

(continues on next page)

(continued from previous page)

```
"""
import numpy as np
import ROOT

# define the number of rows and columns
nrow = 3
ncol = 5

# Initialise and fill a TMatrixD
InMat = ROOT.TMatrixD(nrow, ncol)
for i in range(nrow):
    for j in range(ncol):
        InMat[i][j] = ROOT.gRandom.Gaus(0, 1)

print("Original TMatrixD")
InMat.Print()

# Create the ndarray with the good shape
mat_version = np.frombuffer(InMat.GetMatrixArray(), dtype=np.float64,
                            count=nrow*ncol).reshape(nrow, ncol)
print("Numpy array transformation")
print(mat_version)

# Back to another TMatrixD
OutMat = ROOT.TMatrixD(nrow, ncol)
OutMat.SetMatrixArray(mat_version)
print("\nRecreated TMatrixD")
OutMat.Print()
```

The macro shown above, is an example of how to convert the content stored in a `TMatrixD` but also how to go from a `numpy.array` back to `TMatrixD` if one needs to provide this to an Uranie's method. Its result is discussed in *Macro "howtoConvertTMatrixDArray.py"*.

1.3.5.3 Use a C++ function interactively

This section describes how to use a C++ function within the python framework. The idea is that every of our surrogate model discussed later-on on can be exported in C++ and so they can be used either throughout the Uranie classes (for instance to evaluate a set of new points) or, for single evaluation, in either C++ or python. This macro will focus on the latter part. This macro is also detailled *Macro "howtoLoadFunction.py"*.

```
"""
Example of C++ function in python
"""
import numpy as np
import ROOT

# Loading a function
ROOT.gROOT.LoadMacro("UserFunctions.C")

# Preparing inputs and outputs
inp = np.array([1.2, 0.8])  # input values
out = np.zeros(4)  # output values initialise with 4 zeros
```

(continues on next page)

(continued from previous page)

```
# Call the method, through the ROOT interface
ROOT.operation(inp, out)

# Print the result
print("Results are")
print(out)
```

The macro shown above, is an example of how to load a function and use it in a single-point estimation. Its result is discussed in *Macro "howtoLoadFunction.py"*.

1.3.5.4 Pass argument "by-reference"

This sections shows how to use function that would have been defined using the "by-reference" prototype, which is specific to C++. This macro is also detailled *Macro "howtoPassReference.py"*.

```
"""
Example of C++style reference usage
"""
from ctypes import c_double  # for ROOT version greater or equal to 6.20
from URANIE import DataServer

# create a dataserver and read data
tds = DataServer.TDataServer("pouet", "foo")
tds.fileDataRead("myData.dat")

# compute quantile (this method get the results 'by reference')
proba = 0.9  # ROOT.Double(0.9) for ROOT version lower than 6.20
quant = c_double(0.0)  # ROOT.Double(0.0) for ROOT version lower than 6.20

# Compute the quantile and dump the value returned by reference
tds.computeQuantile("x", proba, quant)
print("Result is ")
print(quant.value)
```

The macro shown above, is an example of how to load a function and use it in a single-point estimation. Its result is discussed in *Macro "howtoPassReference.py"*.

1.3.6 References

The PyROOT environment has a few specificities which it is preferable to be aware of (the rest being discussed already in *References*). 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 [Gil09] introduces (in French) some of these problems and shows examples on how to use Uranie with PyROOT.

CHAPTER

TWO

THE DATASERVER MODULE

Abstract This chapter is dealing with the DataServer module which is the spine of the Uranie project. In this module crucial classes are defined to handle data and transmit them to other modules, along with basic statistics implementation.

The source files are in **souRCE.git/dataSERVER/souRCE** and the corresponding namespace is **URANIE::DataServer**.

2.1 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 1.1). The already discussed `TDataServer` contains two main objects (has shown in Figure 2.1):

- 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 *The TAttribute class*.
- 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.

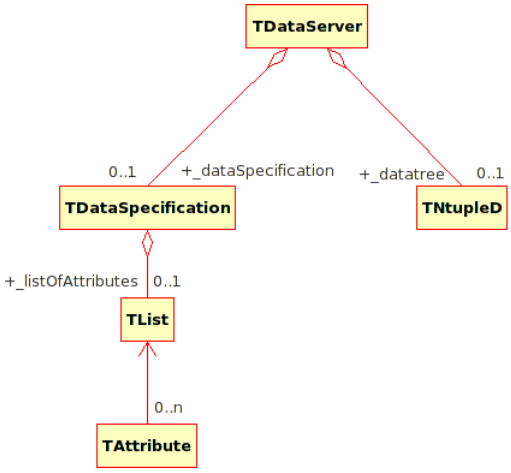


Figure 2.1: Diagram of the class `TDataServer`

The chapter presentation will be articulated as follows. First of all, the way variable are handled will be introduced in *The TAttribute class*. 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 *Data handling* before discussing the basic statistical treatment (in *Statistical treatments and operations*) and the dedicated visualisation tool (in *Visualisation dedicated to uncertainties*).

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")
```

2.2 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 *Statistical treatments and operations*

2.2.1 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::EType` 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.

2.2.2 List of variable information

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

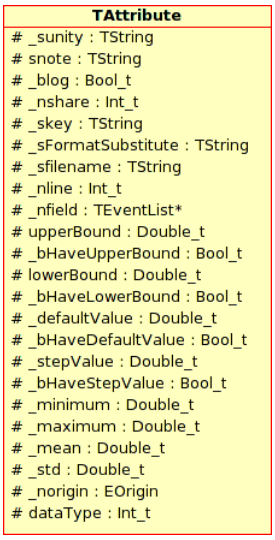


Figure 2.2: Attributes of TAttribute class

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 *Computing the elementary statistic*

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 *The quantile computation*

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:

```
px = DataServer.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

```
psdp = DataServer.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("sdp", "x2", "#sigma_{#Delta P}", "M^{2}")
tdsGeyser.draw("sdp")
```

The result of this piece of code is shown in Figure 2.3.

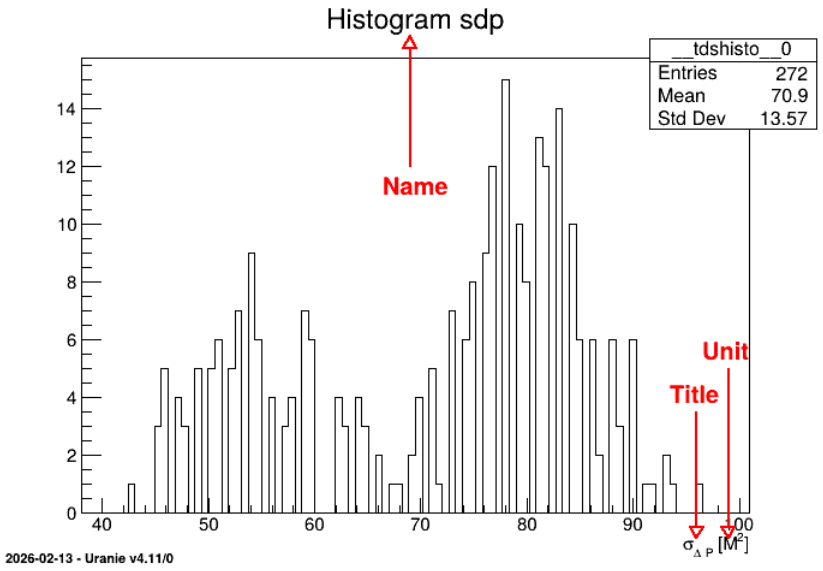


Figure 2.3: Graph of the variable `sdp`

- 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

```
x = DataServer.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

```
xvec = DataServer.TAttribute("x", 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:

```
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 2.4.

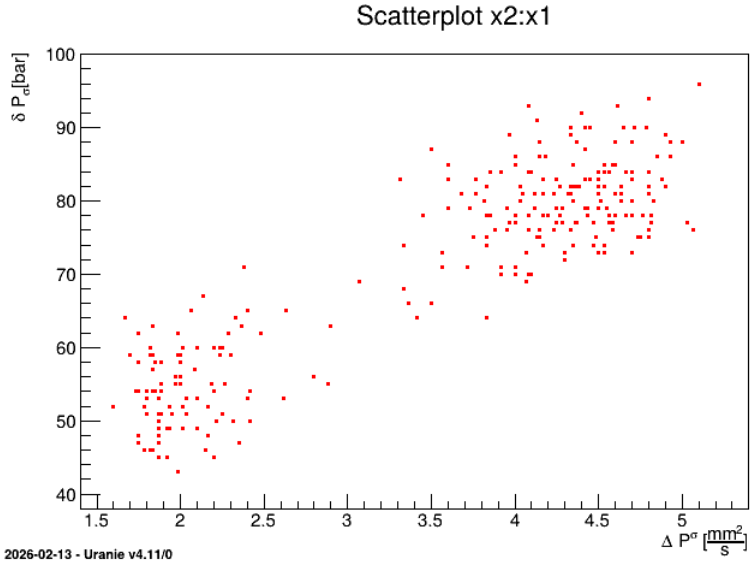


Figure 2.4: Scatterplot `x2` versus `x1` for the `geyser` data with modification of fields `title` and `unit`

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

```
real = DataServer.TAttribute("real")
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

```
import numpy as np

vector = DataServer.TAttribute("vector", DataServer.TAttribute.kVector)
v_value = ROOT.std.vector("double")([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

```
string = DataServer.TAttribute("string", DataServer.TAttribute.kString)
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 *Code input and output files*.

- **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 *The Launcher module*.

2.2.3 Examples of use of the class `TAttribute`

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

```
px = DataServer.TAttribute("x")
```

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

```
px = DataServer.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:

```
px = DataServer.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

```
px = DataServer.TAttribute("x", DataServer.TAttribute.kString)
```

2.2.4 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:

```
tds = DataServer.TDataServer("foo","pouet")
tds.fileDataRead("geyser.dat")
# Adding a new attribute
x3 = DataServer.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++:

```
import numpy as np

tds = DataServer.TDataServer("foo","tru")
tds.fileDataRead("myData.dat")
# Defining a vector with 11 elements
x2 = np.array([-10,-8,-6,-4,-2,0,2,4,6,8,10], dtype=np.float64)
# Call the method using the address of first element and the size of it
tds.addAttributeUsingData("x2", x2, len(x2))
```

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.

2.2.5 Introducing the `TStochasticAttribute` classes

The `TStochasticAttribute` is the parent class to all attributes which values can be generated by a `TSampler` (as discussed in *The Stochastic methods*). 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 *Composing law*, leads to a new probability density function that would look like

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

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 [App13].

Table 2.1 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).

Table 2.1: List of Uranie classes representing the probability laws

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)	

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:

```
# Uniform law
pxu = DataServer.TUniformDistribution("x1", -1.0 , 1.0) # (1)
# Gaussian Law
↪
pxn = DataServer.TNormalDistribution("x2", -1.0 , 1.0) # (2)
```

1. Allocation of a pointer *pxu* to a random uniform variable *x1* in interval [-1.0, 1.0].
2. Allocation of a pointer *pxn* to a random normal variable *x2* with mean value $\mu = -1.0$ and standard deviation $\sigma = 1.0$.

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 *Uniform Law* until *InvGamma law*. Here is a brief description of the probability density functions and their parameters.

[1] for a definition of PDF (*probability density function*), CDF (*cumulative density function*) and inverse CDF, please look at [Bla17]

2.2.5.1 Uniform Law

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

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

Uranie code to simulate an uniform random variable is:

```
tds = DataServer.TDataServer("tdssampler", "Sampler Uranie demo")
tds.addAttribute(DataServer.TUniformDistribution("u", -2., 3.))

fsamp = Sampler.TSampling(tds, "lhs", 300)
fsamp.generateSample()  # Create a representative sample

tds.Draw("u")
```

Figure 2.5 shows the PDF, CDF and inverse CDF generated for a given set of parameters.

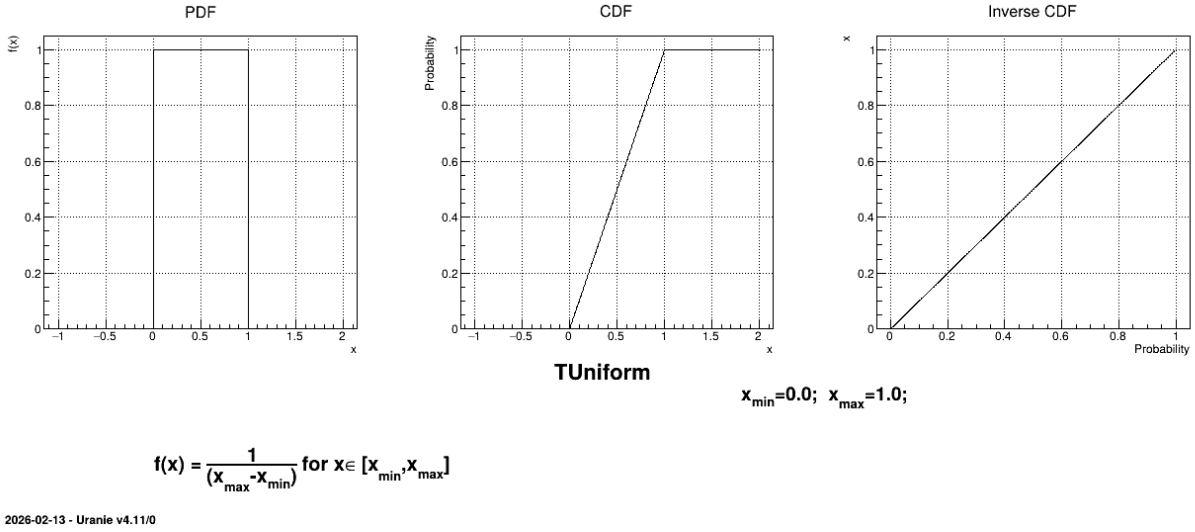


Figure 2.5: Example of PDF, CDF and inverse CDF for Uniform distributions.

2.2.5.2 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

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

Uranie code to simulate a LogUniform random variable is:

```
tds = DataServer.TDataServer("tdssampler", "Sampler Uranie demo")
tds.addAttribute(DataServer.TLogUniformDistribution("lu", .001, 10.))

fsamp = Sampler.TSampling(tds, "lhs", 300)
fsamp.generateSample()  # Create a representative sample
```

(continues on next page)

(continued from previous page)

```
tds.Draw("lu")
tds.Draw("log(lu)")  # Check that ln(x) follows a uniform law
```

Figure 2.6 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

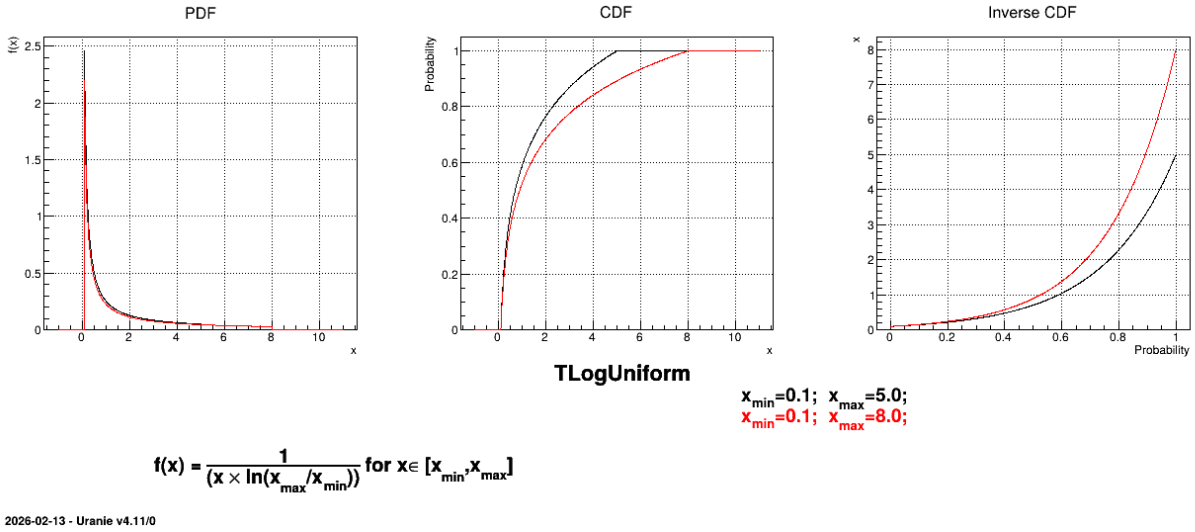


Figure 2.6: Example of PDF, CDF and inverse CDF for LogUniform distributions.

2.2.5.3 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

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

and

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

Uranie code to simulate a triangular random variable is:

```
tds = DataServer.TDataServer("tdssampler", "Sampler Uranie demo")
tds.addAttribute(DataServer.TTriangularDistribution("t", 5.0, 8., 6.0))

fsamp = Sampler.TSampling(tds, "lhs", 300)
fsamp.generateSample()  # Create a representative sample

tds.Draw("t")
```

Figure 2.7 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

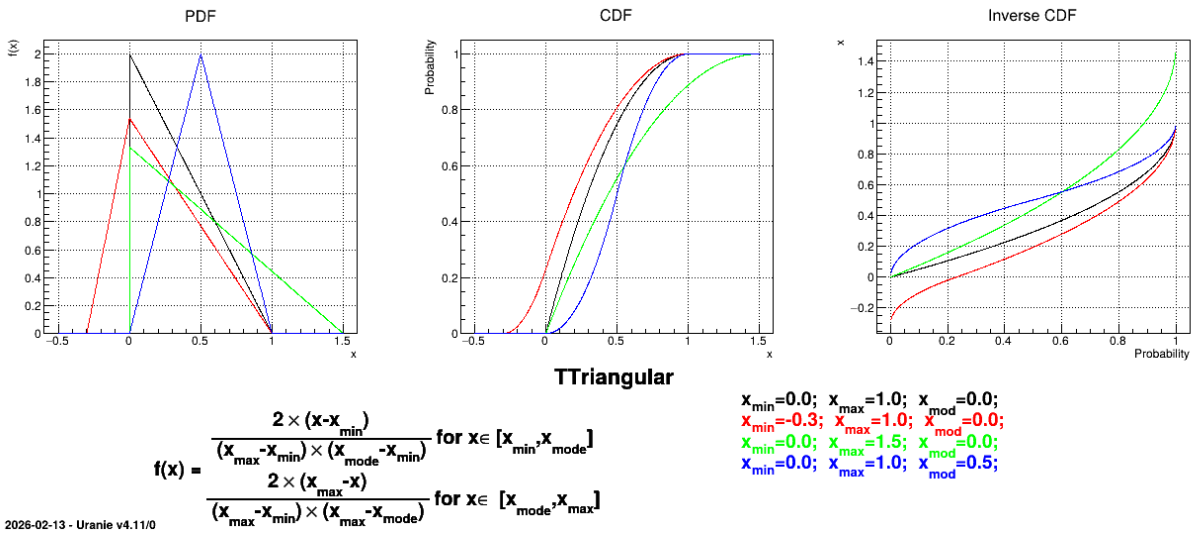


Figure 2.7: Example of PDF, CDF and inverse CDF for Triangular distributions.

2.2.5.4 LogTriangular Law

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

$$f(x) = \frac{2 \times \ln(x/x_{\min})}{x \times \ln(x_{\max}/x_{\min}) \times \ln(x_{\text{mode}}/x_{\min})} \mathbb{1}_{[x_{\min}, x_{\text{mode}}]}(x)$$

and

$$f(x) = \frac{2 \times \ln(x_{\max}/x)}{x \times \ln(x_{\max}/x_{\min}) \times \ln(x_{\max}/x_{\text{mode}})} \mathbb{1}_{[x_{\text{mode}}, x_{\max}]}(x)$$

Uranie code to simulate a LogTriangular random variable is:

```
tds = DataServer.TDataServer("tdssampler", "Sampler Uranie demo")
tds.addAttribute(DataServer.TLogTriangularDistribution("lt", .001, 10., 2.5))

fsamp = Sampler.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 2.8 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

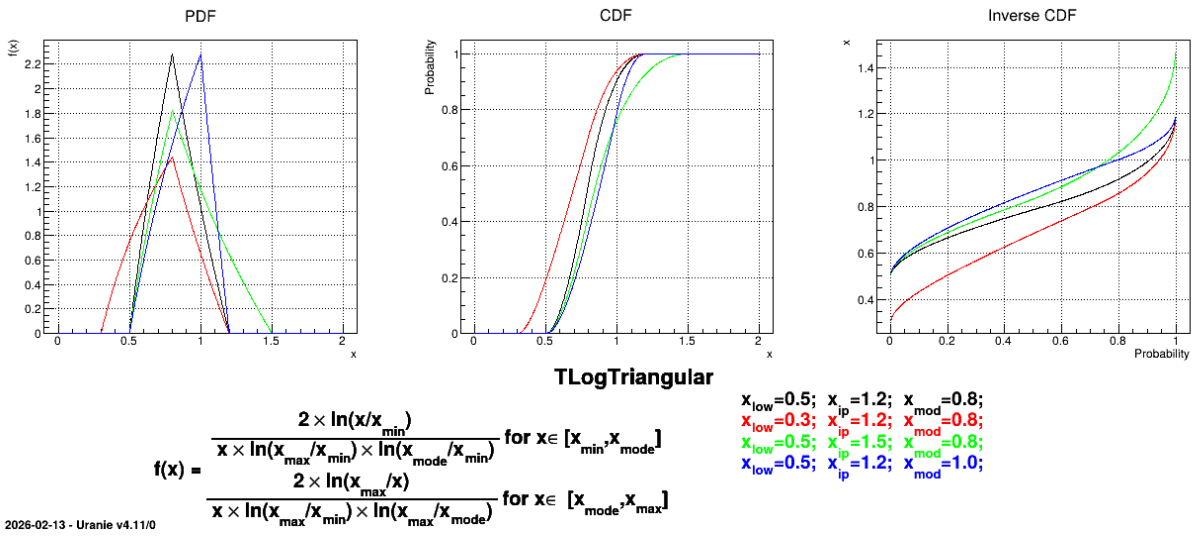


Figure 2.8: Example of PDF, CDF and inverse CDF for LogTriangular distributions.

2.2.5.5 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:

```
tds = DataServer.TDataServer("tdssampler", "Sampler Uranie demo")
tds.addAttribute(DataServer.TNormalDistribution("n", 0.0, 1.0))

fsamp = Sampler.TSampling(tds, "lhs", 300)
fsamp.generateSample()  # Create a representative sample

tds.Draw("n")
```

Figure 2.9 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

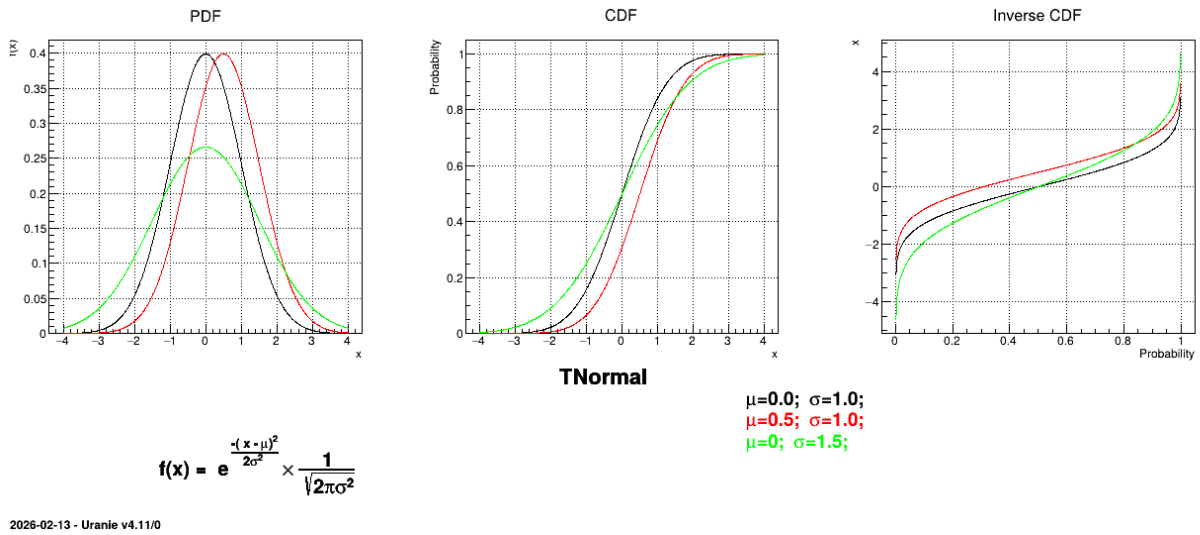


Figure 2.9: Example of PDF, CDF and inverse CDF for Normal distributions.

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 2.10 for a given set of parameters and various boundaries.

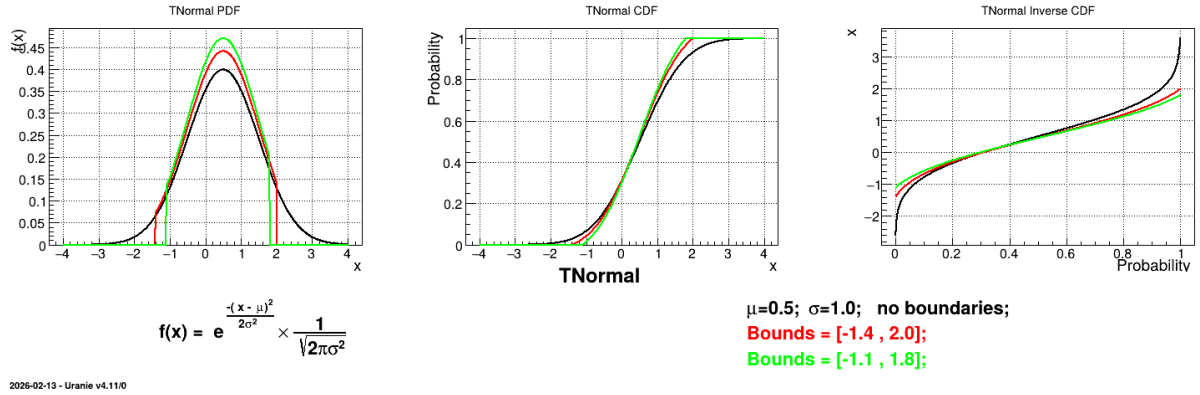


Figure 2.10: Example of PDF, CDF and inverse CDF for a Normal truncated distribution.

2.2.5.6 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

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

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

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

Uranie code to simulate a LogNormal random variable is:

```
tds = DataServer.TDataServer("tdssampler", "Sampler Uranie demo")
# using M, Ef and xmin
tds.addAttribute(DataServer.TLogNormalDistribution("ln", 1.2, 1.5, -0.5))
# to use ln(x) properties:
# mu = 0.5
# sigma = 1
# tds.setUnderlyingNormalParameters(mu,sigma)

fsamp = Sampler.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 2.11 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

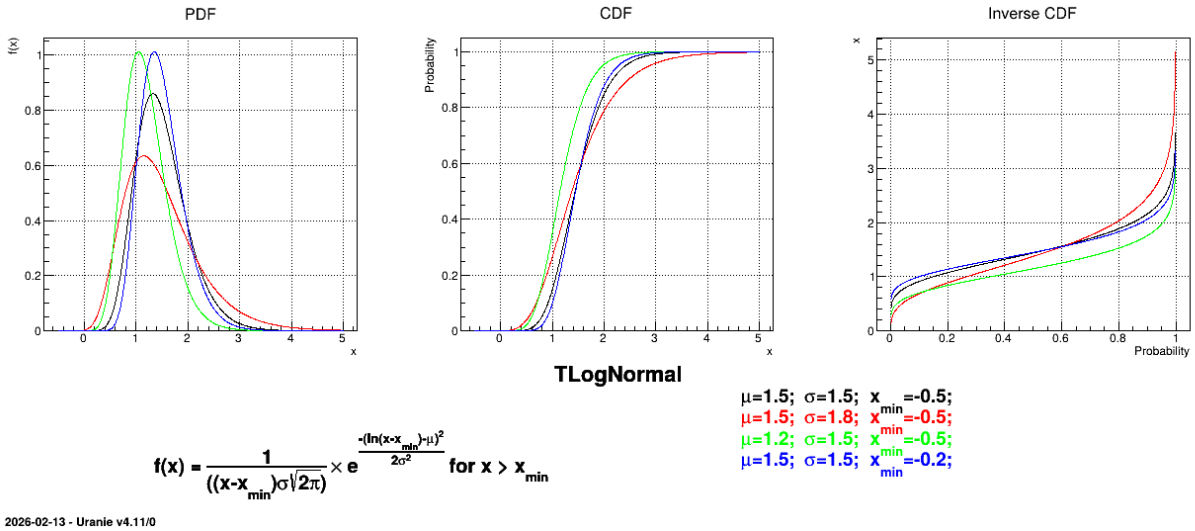


Figure 2.11: Example of PDF, CDF and inverse CDF for LogNormal distributions.

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 2.12 for a given set of parameters and various boundaries.

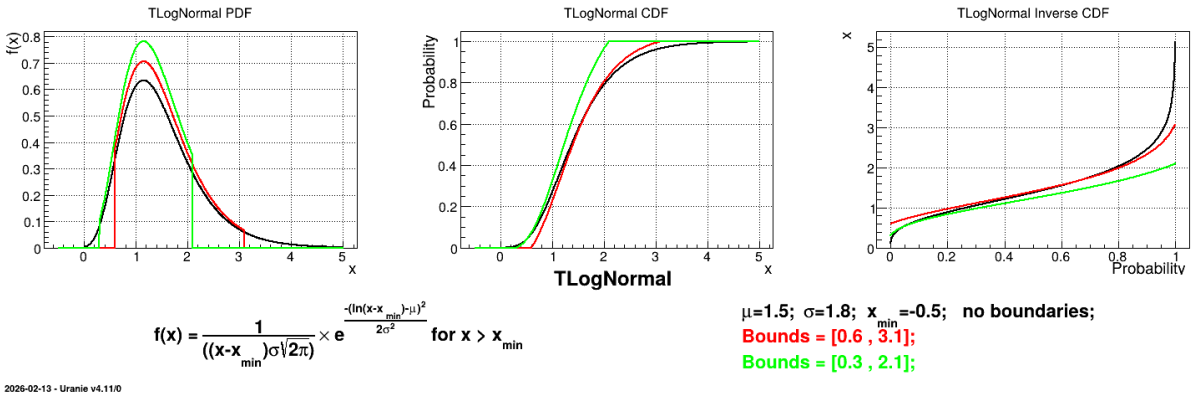


Figure 2.12: Example of PDF, CDF and inverse CDF for a LogNormal truncated distribution.

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

Uranie code to simulate a Trapezium random variable is:

```
tds = DataServer.TDataServer("tdssampler", "Sampler Uranie demo")
tds.addAttribute(DataServer.TTrapeziumDistribution("tr", 0.0, 1.0, 0.25, 0.75) )

fsamp = Sampler.TSampling(tds, "lhs", 300)
fsamp.generateSample()  # Create a representative sample

tds.Draw("tr")
```

Figure 2.13 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

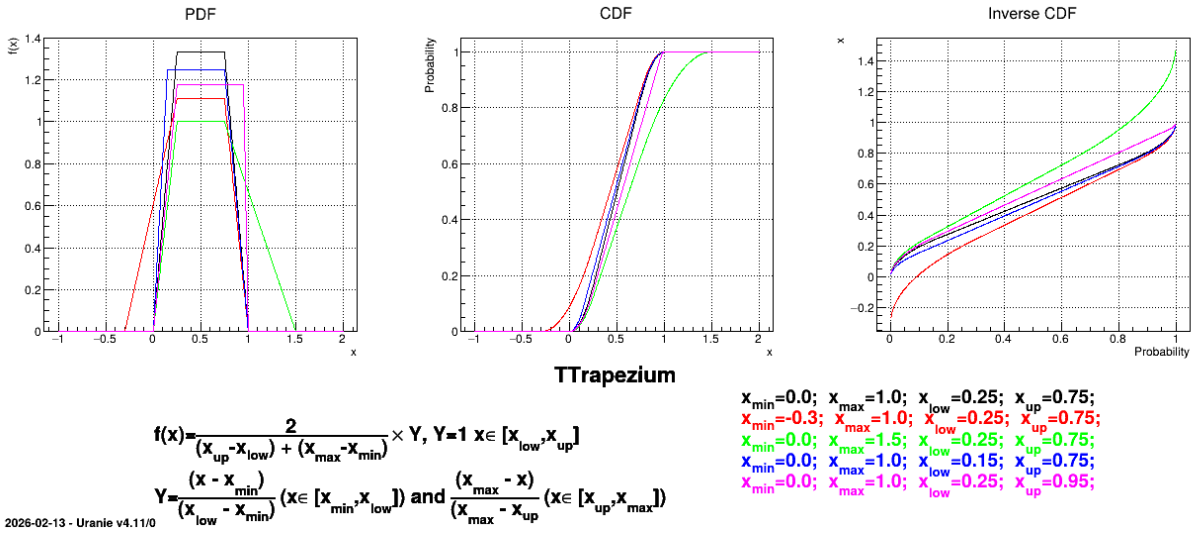


Figure 2.13: Example of PDF, CDF and inverse CDF for Trapezium distributions.

2.2.5.8 UniformByParts law

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

$$f(x) = \frac{0.5}{(x_{\text{med}} - x_{\text{min}})} \mathbb{1}_{[x_{\text{min}}, x_{\text{med}}]}(x) \qquad \text{and} \qquad f(x) = \frac{0.5}{(x_{\text{max}} - x_{\text{med}})} \mathbb{1}_{[x_{\text{med}}, x_{\text{max}}]}(x)$$

Uranie code to simulate a UniformByParts random variable is:

```
tds = DataServer.TDataServer("tdssampler", "Sampler Uranie demo")
tds.addAttribute(DataServer.TUniformByPartsDistribution("ubp", 0.0, 1.0, 0.5) )

fsamp = Sampler.TSampling(tds, "lhs", 300)
fsamp.generateSample()  # Create a representative sample

tds.Draw("ubp")
```

Figure 2.14 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

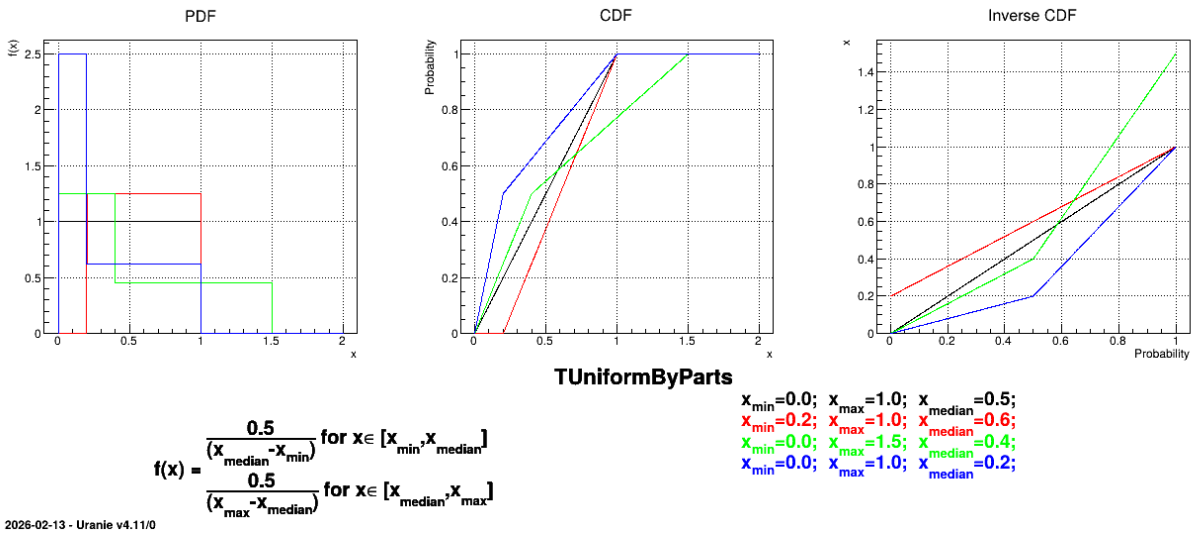


Figure 2.14: Example of PDF, CDF and inverse CDF for UniformByParts distributions.

2.2.5.9 Exponential law

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

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

The rate parameter λ should be greater than 0.0001.

Uranie code to simulate an Exponential random variable is:

```
tds = DataServer.TDataServer("tdssampler", "Sampler Uranie demo")
tds.addAttribute(DataServer.TExponentialDistribution("exp", 0.5))

fsamp = Sampler.TSampling(tds, "lhs", 300)
fsamp.generateSample()  # Create a representative sample

tds.Draw("exp")
```

Figure 2.15 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

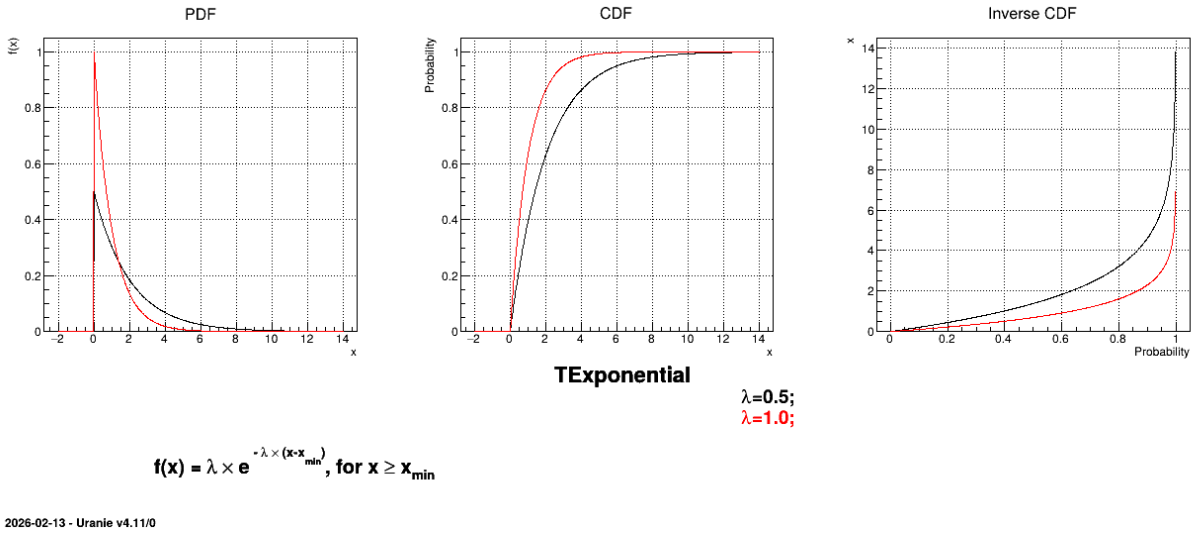


Figure 2.15: Example of PDF, CDF and inverse CDF for Exponential distributions.

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 2.16 for a given set of parameters and various boundaries.

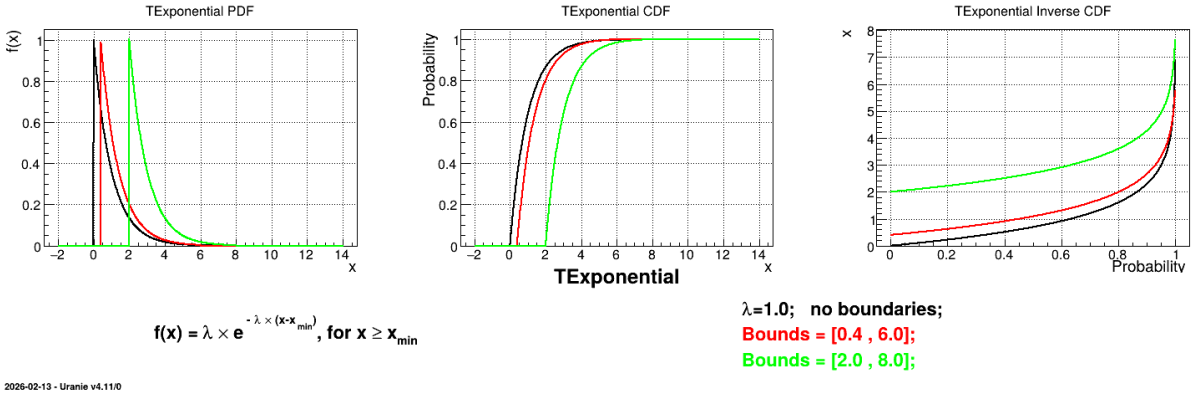


Figure 2.16: Example of PDF, CDF and inverse CDF for a Exponential truncated distribution.

2.2.5.10 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:

```
tds = DataServer.TDataServer("tdssampler", "Sampler Uranie demo")
tds.addAttribute(DataServer.TCauchyDistribution("cau", 0.3, 1.0))

fsamp = Sampler.TSampling(tds, "lhs", 300)
fsamp.generateSample()  # Create a representative sample

tds.Draw("cau")
```

Figure 2.17 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

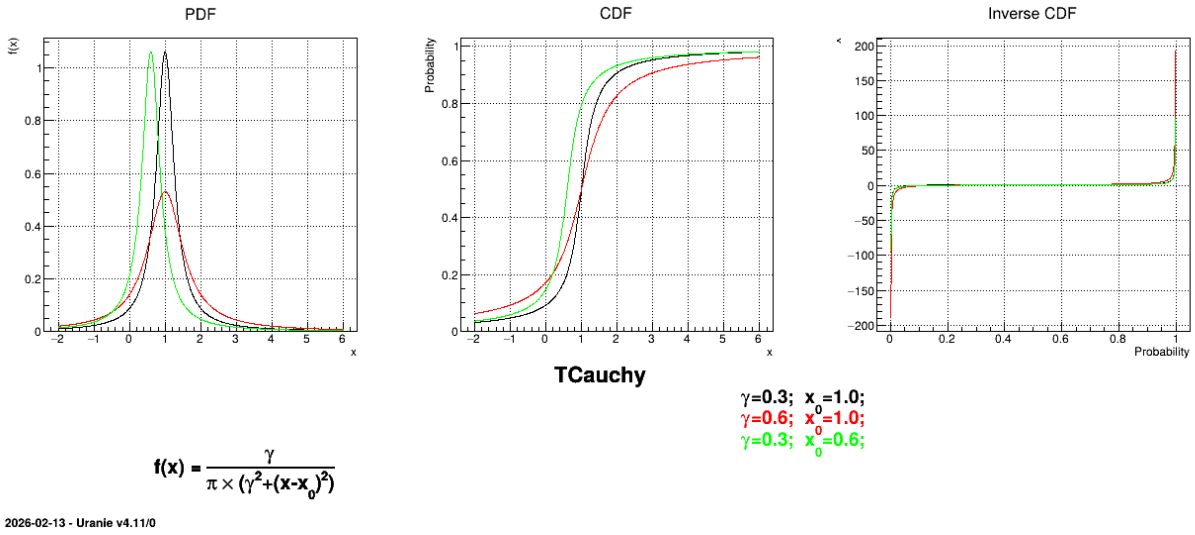


Figure 2.17: Example of PDF, CDF and inverse CDF for Cauchy distributions.

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 2.18 for a given set of parameters and various boundaries.

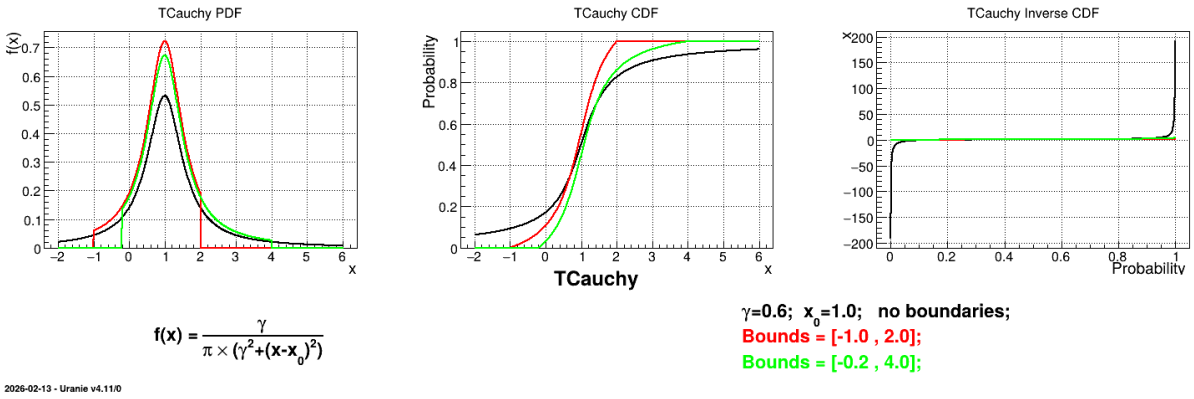


Figure 2.18: Example of PDF, CDF and inverse CDF for a Cauchy truncated distribution.

2.2.5.11 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:

```
tds = DataServer.TDataServer("tdssampler", "Sampler Uranie demo")
tds.addAttribute(DataServer.TGumbelMaxDistribution("gm", 0.5, 2.0))

fsamp = Sampler.TSampling(tds, "lhs", 300)
fsamp.generateSample()  # Create a representative sample

tds.Draw("gm")
```

Figure 2.19 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

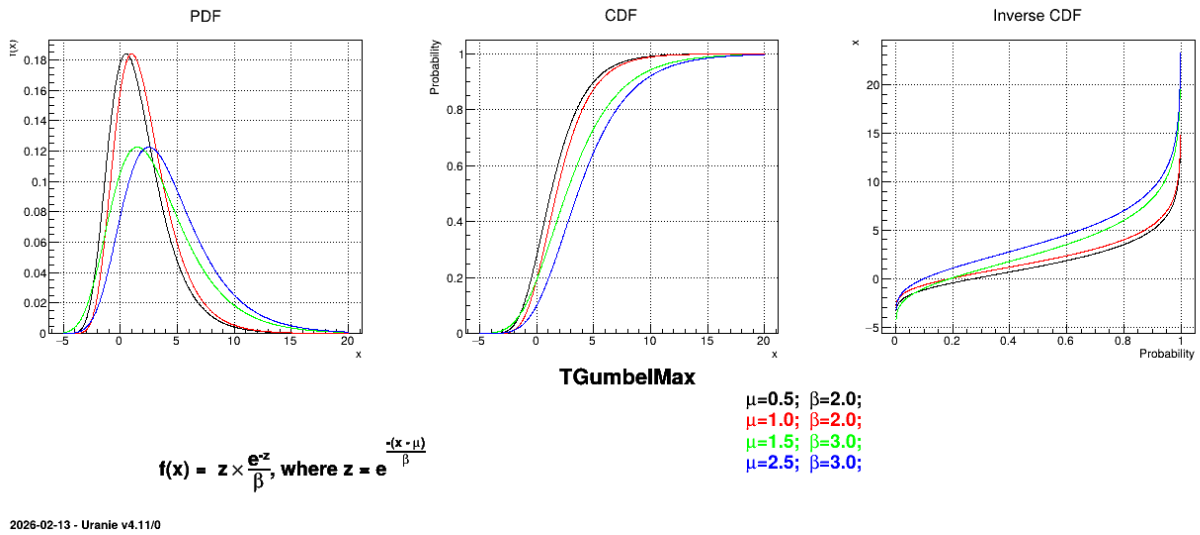


Figure 2.19: Example of PDF, CDF and inverse CDF for GumbelMax distributions.

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 2.20 for a given set of parameters and various boundaries.

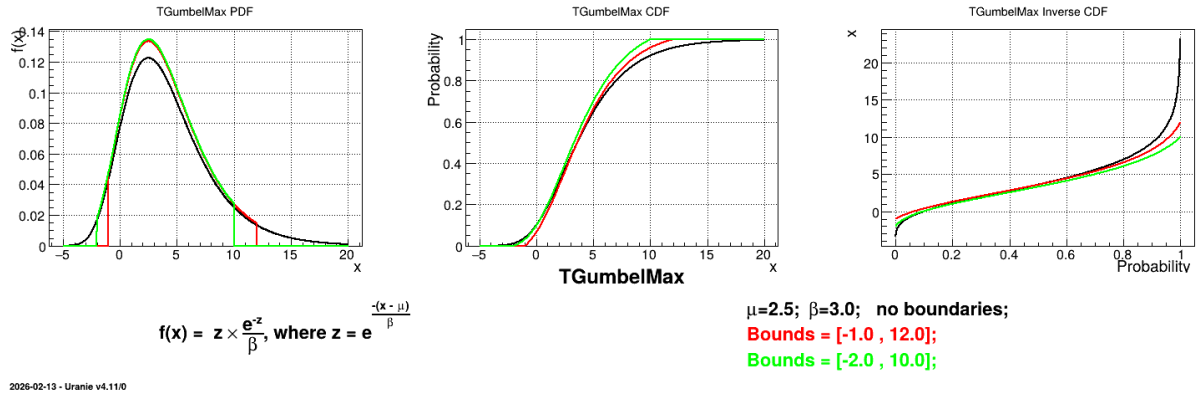


Figure 2.20: Example of PDF, CDF and inverse CDF for a GumbelMax truncated distribution.

2.2.5.12 Weibull law

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

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

Both λ and k should be greater than 0.0001.

Uranie code to simulate a Weibull random variable is:

```
tds = DataServer.TDataServer("tdssampler", "Sampler Uranie demo")
tds.addAttribute(DataServer.TWeibullDistribution("wei", 0.5, 2.0, -0.01) )

fsamp = Sampler.TSampling(tds, "lhs", 300)
fsamp.generateSample()  # Create a representative sample

tds.Draw("wei")
```

Figure 2.21 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

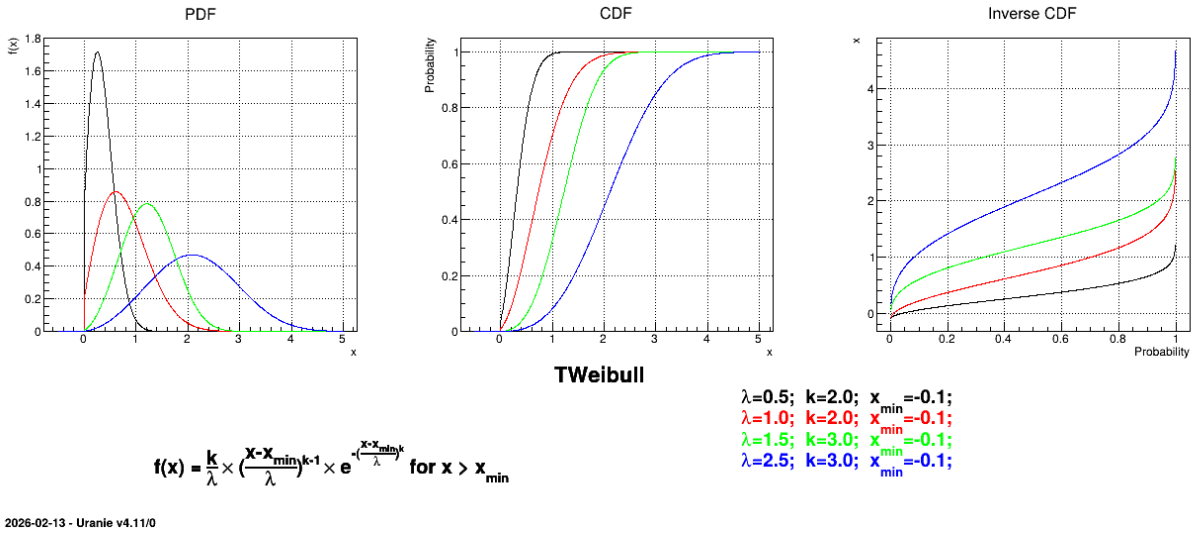


Figure 2.21: Example of PDF, CDF and inverse CDF for Weibull distributions.

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 2.22 for a given set of parameters and various boundaries.

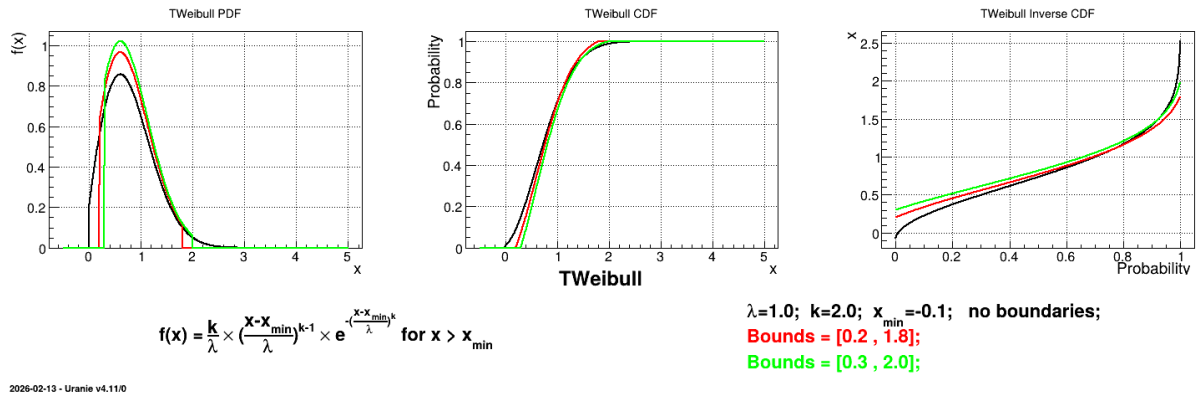


Figure 2.22: Example of PDF, CDF and inverse CDF for a Weibull truncated distribution.

2.2.5.13 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:

```
tds = DataServer.TDataServer("tdssampler", "Sampler Uranie demo")
tds.addAttribute(DataServer.TBetaDistribution("bet", 6.0, 6.0, 0.0, 2.0) )

fsamp = Sampler.TSampling(tds, "lhs", 300)
fsamp.generateSample()  # Create a representative sample

tds.Draw("bet")
```

Figure 2.23 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

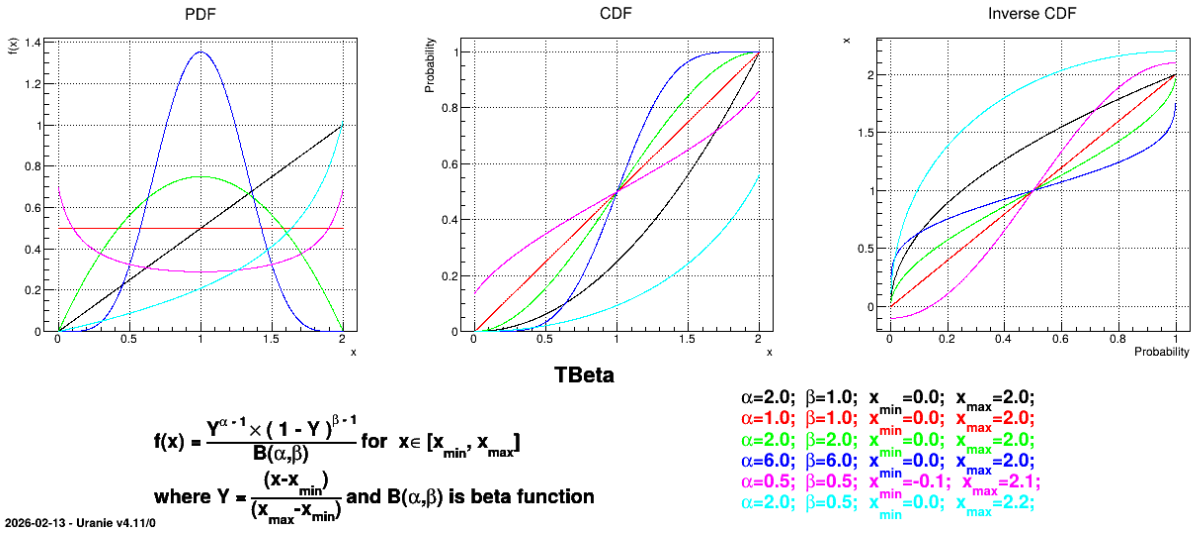


Figure 2.23: Example of PDF, CDF and inverse CDF for Beta distributions.

2.2.5.14 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:

```
tds = DataServer.TDataServer("tdssampler", "Sampler Uranie demo")
tds.addAttribute(DataServer.TGenParetoDistribution("gpa", 1.0, 1.0, 0.3))

fsamp = Sampler.TSampling(tds, "lhs", 300)
fsamp.generateSample()  # Create a representative sample

tds.Draw("gpa")
```

Figure 2.24 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

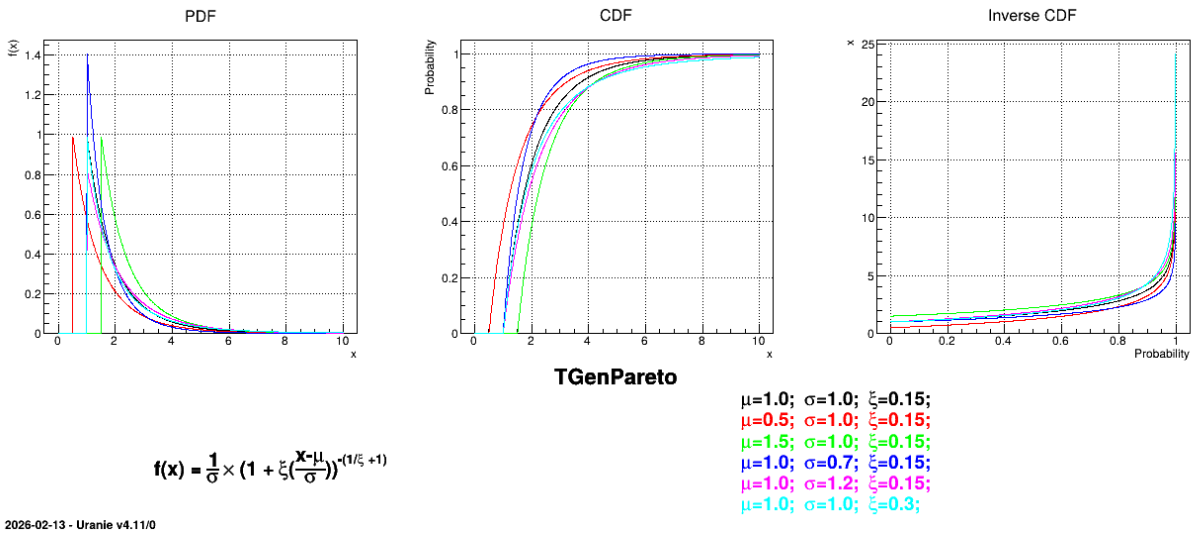


Figure 2.24: Example of PDF, CDF and inverse CDF for GenPareto distributions.

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 2.25 for a given set of parameters and various boundaries.

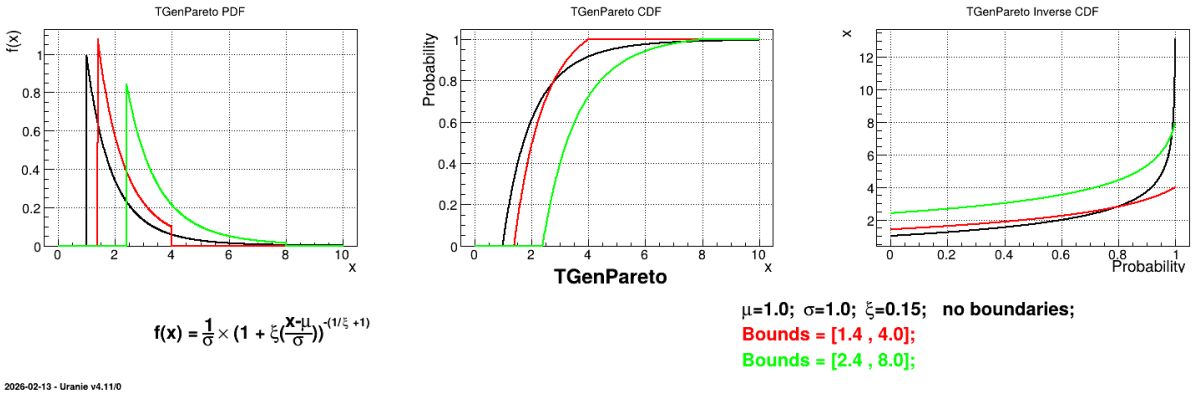


Figure 2.25: Example of PDF, CDF and inverse CDF for a GenPareto truncated distribution.

2.2.5.15 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:

```
tds = DataServer.TDataServer("tdssampler", "Sampler Uranie demo")
tds.addAttribute(DataServer.TGammaDistribution("gam", 1.0, 2.0, 0.0))

fsamp = Sampler.TSampling(tds, "lhs", 300)
fsamp.generateSample()  # Create a representative sample

tds.Draw("gam")
```

Figure 2.26 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

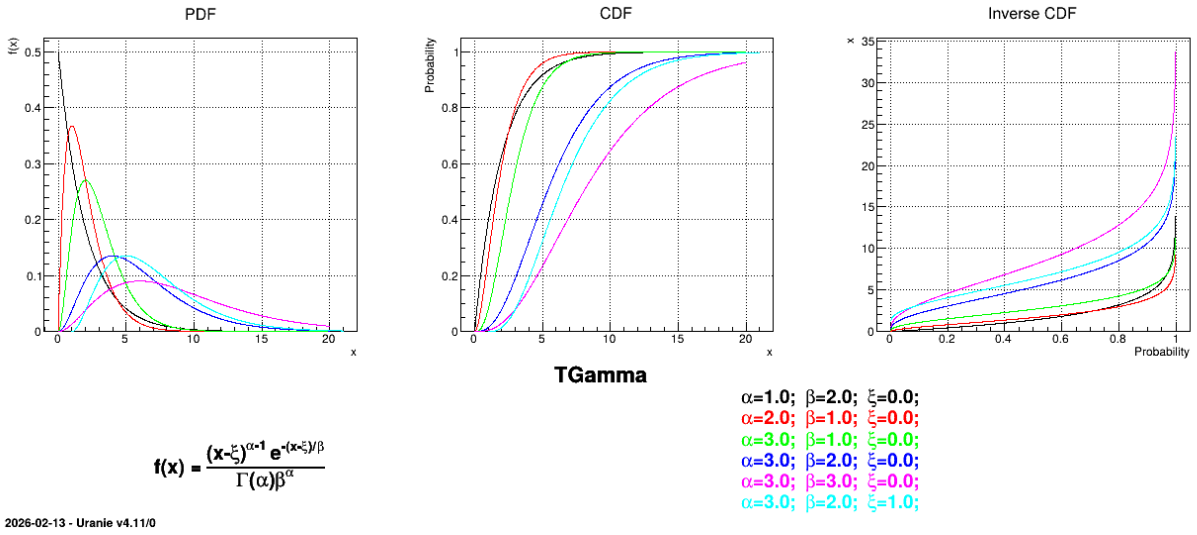


Figure 2.26: Example of PDF, CDF and inverse CDF for Gamma distributions.

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 2.27 for a given set of parameters and various boundaries.

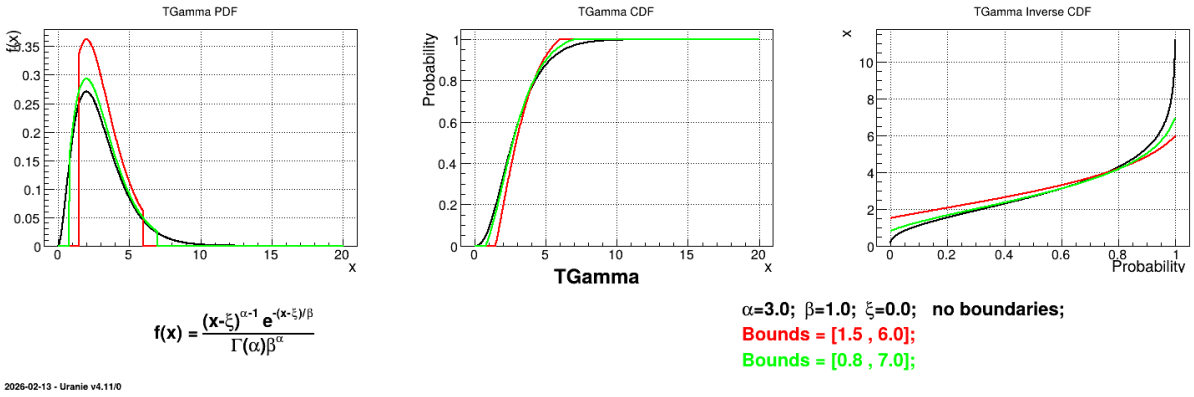


Figure 2.27: Example of PDF, CDF and inverse CDF for a Gamma truncated distribution.

2.2.5.16 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.

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

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

```
tds = DataServer.TDataServer("tdssampler", "Sampler Uranie demo")
tds.addAttribute(DataServer.TInvGammaDistribution("ing", 2.0, 0.5, 0.0))

fsamp = Sampler.TSampling(tds, "lhs", 300)
fsamp.generateSample()  # Create a representative sample

tds.Draw("ing")
```

Figure 2.28 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

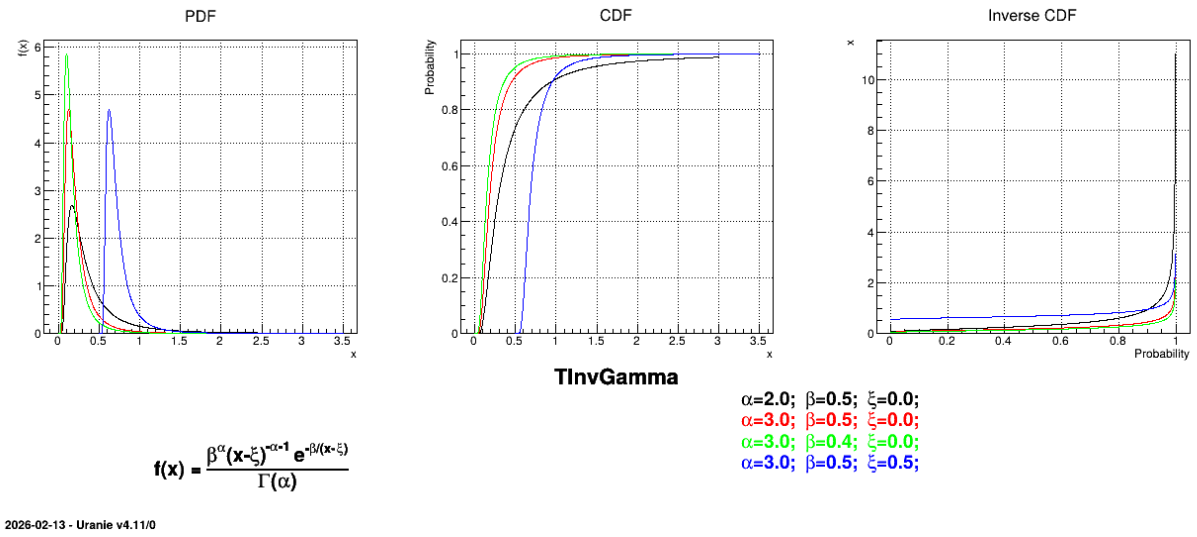


Figure 2.28: Example of PDF, CDF and inverse CDF for InvGamma distributions.

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 2.29 for a given set of parameters and various boundaries.

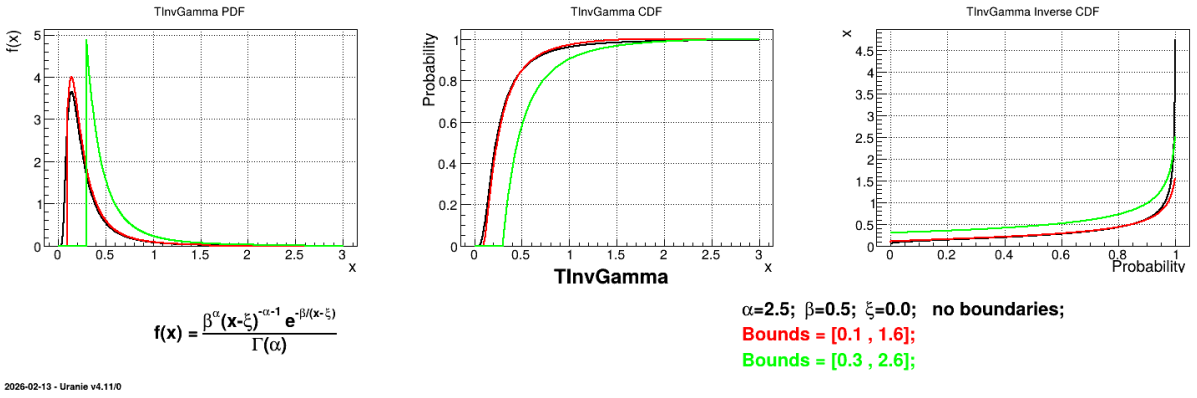


Figure 2.29: Example of PDF, CDF and inverse CDF for a InvGamma truncated distribution.

2.2.5.17 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:

```
tds = DataServer.TDataServer("tdssampler", "Sampler Uranie demo")
tds.addAttribute(DataServer.TStudentDistribution("stu", 5))

fsamp = Sampler.TSampling(tds, "lhs", 300)
fsamp.generateSample()  # Create a representative sample

tds.Draw("stu")
```

Figure 2.30 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

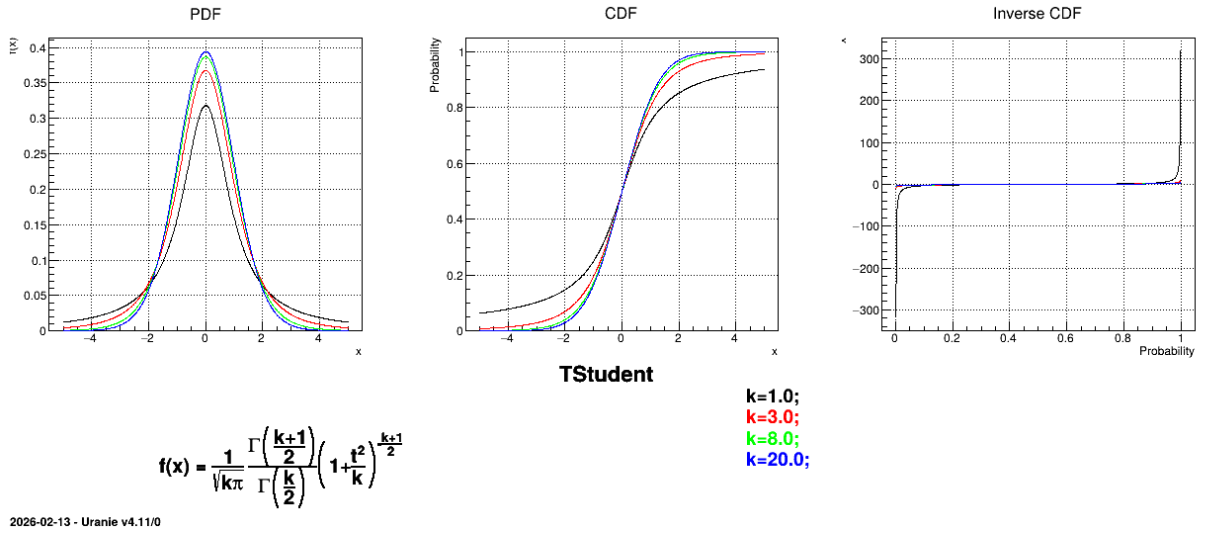


Figure 2.30: Example of PDF, CDF and inverse CDF for Student distributions.

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 2.31 for a given set of parameters and various boundaries.

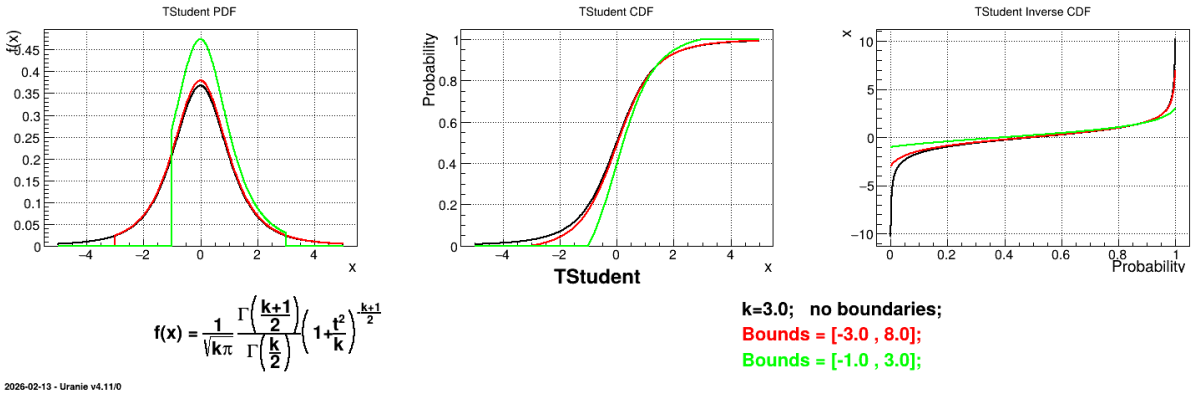


Figure 2.31: Example of PDF, CDF and inverse CDF for a Student truncated distribution.

2.2.5.18 Generalized normal law

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

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

Both α and β should be greater than 0.

Uranie code to simulate a generalized normal random variable is:

```
tds = DataServer.TDataServer("tdssampler", "Sampler Uranie demo")
tds.addAttribute(DataServer.TGeneralizedNormalDistribution("gennor", 0.0, 1.0, 3.0) )

fsamp = Sampler.TSampling(tds, "lhs", 300)
fsamp.generateSample()  # Create a representative sample

tds.Draw("gennor")
```

Figure 2.32 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

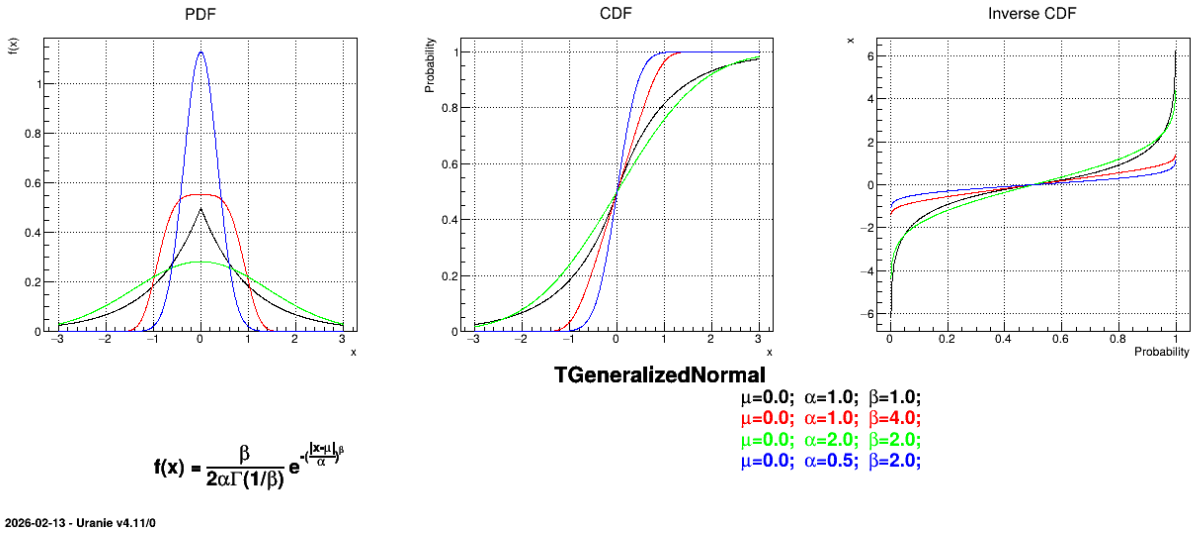


Figure 2.32: Example of PDF, CDF and inverse CDF for GeneralizedNormal distributions.

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 2.33 for a given set of parameters and various boundaries.

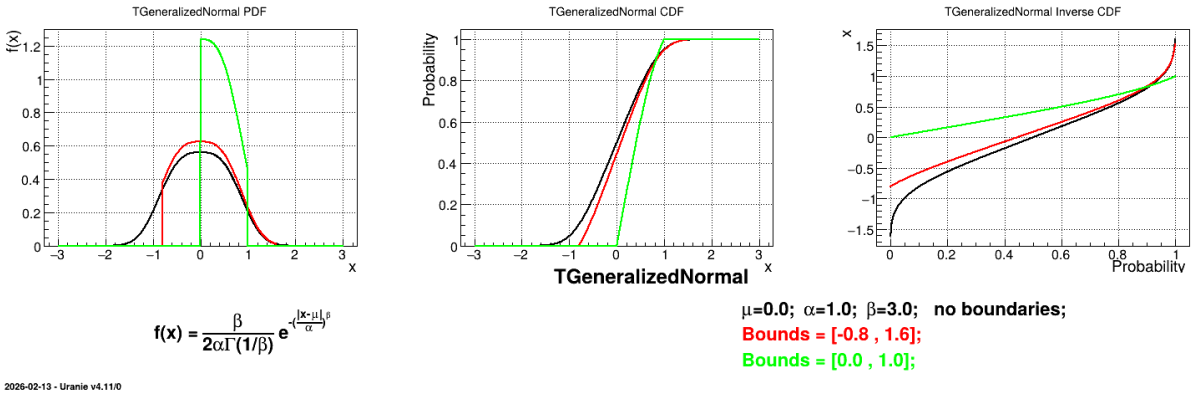


Figure 2.33: Example of PDF, CDF and inverse CDF for a generalized normal truncated distribution.

2.2.5.19 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):

```
tds = DataServer.TDataServer("tdssampler", "Sampler Uranie demo")
comp = DataServer.TComposedDistribution("compo")
comp.addDistribution(DataServer.TNormalDistribution("n1", -1.5, 0.2), 1.2)
comp.addDistribution(DataServer.TNormalDistribution("n2", 0, 0.5), 1.0)
comp.addDistribution(DataServer.TNormalDistribution("n3", 1.5, 0.2), 0.8)
tds.addAttribute(comp)

fsamp = Sampler.TSampling(tds, "lhs", 300)
fsamp.generateSample()  # Create a representative sample

tds.Draw("compo")
```

Figure 2.34 shows the PDF, CDF and inverse CDF generated for different sets of parameters.

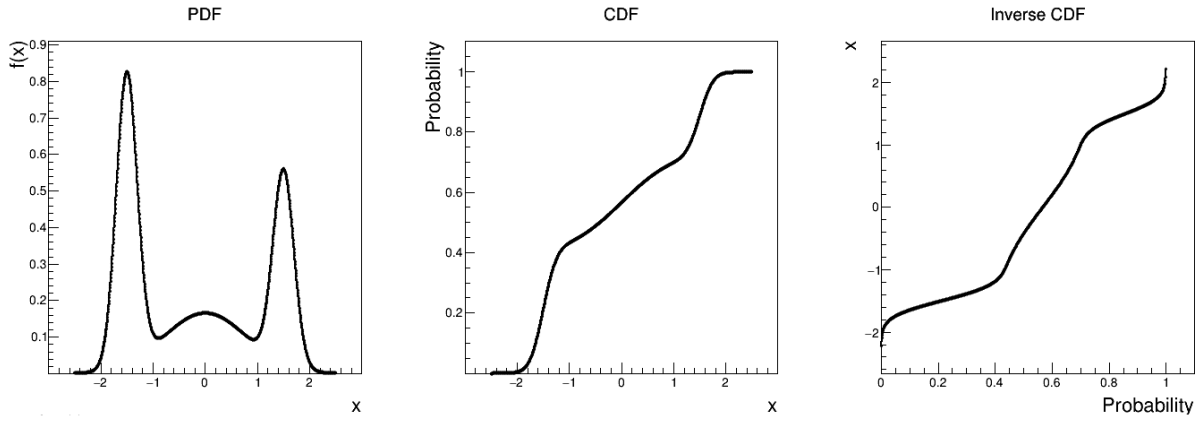


Figure 2.34: Example of PDF, CDF and inverse CDF for a composed distribution made out of three normal distributions with respective weights.

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 2.35 for a given set of parameters and various boundaries.

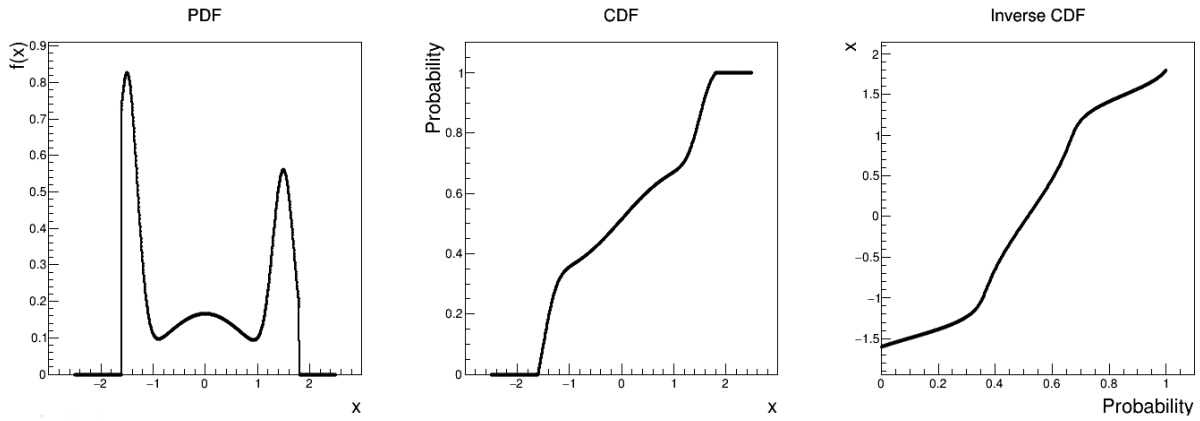


Figure 2.35: Example of PDF, CDF and inverse CDF for a truncated composed distribution made out of three normal distributions with respective weights.

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

```
addDistribution(statt, 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.

2.3 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.

2.3.1 Main format of input/output

2.3.1.1 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:

Table 2.2: List of keywords of *header* in ASCII files.

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

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

```
> 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
```

(continues on next page)

(continued from previous page)

```
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"
```

2.3.1.2 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 2.3.

Table 2.3: List of keywords of *header* in ASCII files.

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

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",
    "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
    }
  ]
}
```

2.3.2 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**

```
from URANIE import DataServer # (1)

tdsGeyser = DataServer.TDataServer("tdsgeyser", "Geyser database") # (2)
tdsGeyser.fileDataRead("geyser.dat") # (3)
tdsGeyser.draw("x2:x1") # (4)
```

Description of the import of an ASCII file

1. Setting the namespace. This instruction is useless when the provided rootlogon has been loaded as all Uranie-namespaces have been loaded as well.
2. 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.
3. Loading data contained in an ASCII file `$URANIESYS/share/uranie/macros/geyser.dat`.
4. Plot of the scatterplot of the variable **x2** versus the variable **x1**

The obtained graph is the following:

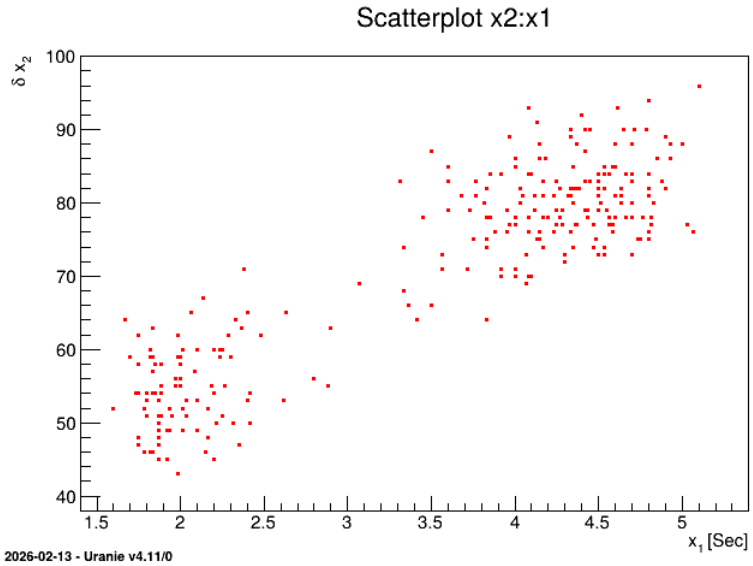


Figure 2.36: Import data from an ASCII file

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 *Macro "dataserverLoadASCIIFilePasture.py"* and *Macro "dataserverLoadASCIIFileCornell.py"*.

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)

2.3.3 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

```
from URANIE import DataServer # (1)

tds = DataServer.TDataServer() # (2)
tds.ntupleDataRead("hsimple.root","ntuple","px*px:*:py*py", "px*px+py*py < 2.0") # (3)
tds.draw("py:px") # (4)
```

Description of data importation of a TTree from a ROOT file

1. Specification of the namespace.
2. Creation of a pointer *tds* to an object of type `TDataServer` without name and title.
3. 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;
4. 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 2.37 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 $px^2 + py^2 < 2$

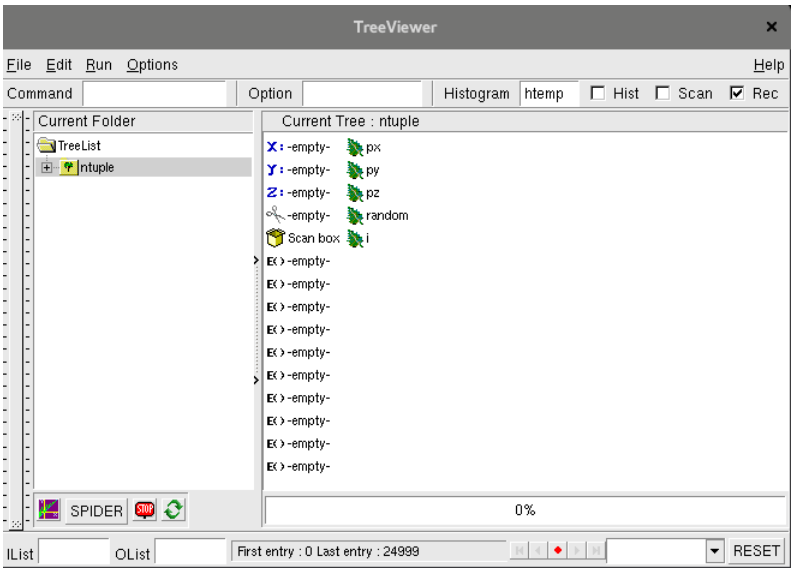


Figure 2.37: Content of the *ntuple* tree contained in “hsimple.root” file.

The obtained graph is as follow:

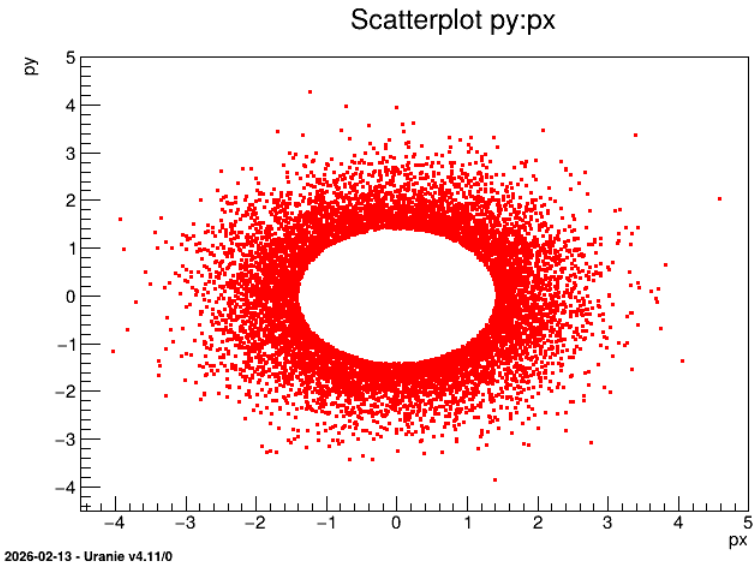


Figure 2.38: Data importation from a TNtuple

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`.

2.3.4 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 *Macro "dataserverAttributes.py"*.

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

```
tds = DataServer.TDataServer("tds", "new TDataServer")
tds.addAttribute(DataServer.TAttribute("x1"))           # (1)
tds.addAttribute(DataServer.TAttribute("x2", 2.5, 5.)) # (2)
tds.addAttribute( "x3" )                                # (3)
tds.addAttribute( "x4", 2.5, 5.)                        # (4)
```

Description of attributes adding to an empty TDataServer

1. Adding a new attribute **x1** to the TDataServer from a `TAttribute` with a name (minimal constructor).
2. Adding a new attribute **x2** to the TDataServer from a `TAttribute` with a name and the extreme values (minimal and maximal).
3. Equivalent to previous one: adding a new attribute **x3** to the TDataServer just by giving its name (minimal constructor).
4. 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 *The TAttribute class*.

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.

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

tdsGeyser.draw("cd2:cd1") # (3)
```

Description attribute adding to a TDataServer from formulas

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

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

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

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

3. Plots the scatterplot of the variable **cd2** versus the variable **cd1**

The obtained graph is:

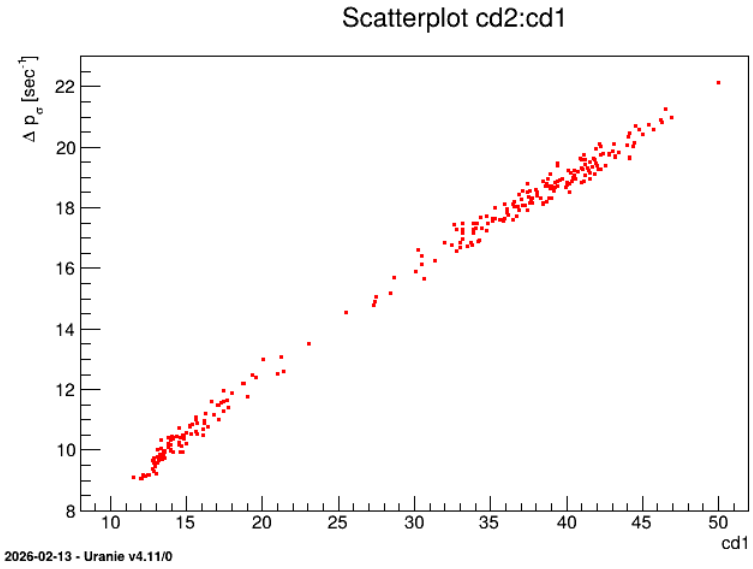


Figure 2.39: Scatterplot of added attributes

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:

```
"""
Example of attribute handling
"""
from URANIE import DataServer

tdsop = DataServer.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
```

(continues on next page)

(continued from previous page)

```
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.

2.3.5 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 *Macro "dataserverMerge.py"*.

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 *Import data from an ASCII file*), the merging being done by calling the `merge` method of the first `TDataServer`

```
"""
Example of data merging (merging two files)
"""
```

(continues on next page)

(continued from previous page)

```
from URANIE import DataServer

tds1 = DataServer.TDataServer()
tds2 = DataServer.TDataServer()

tds1.fileDataRead("tds1.dat")
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.

2.3.6 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:

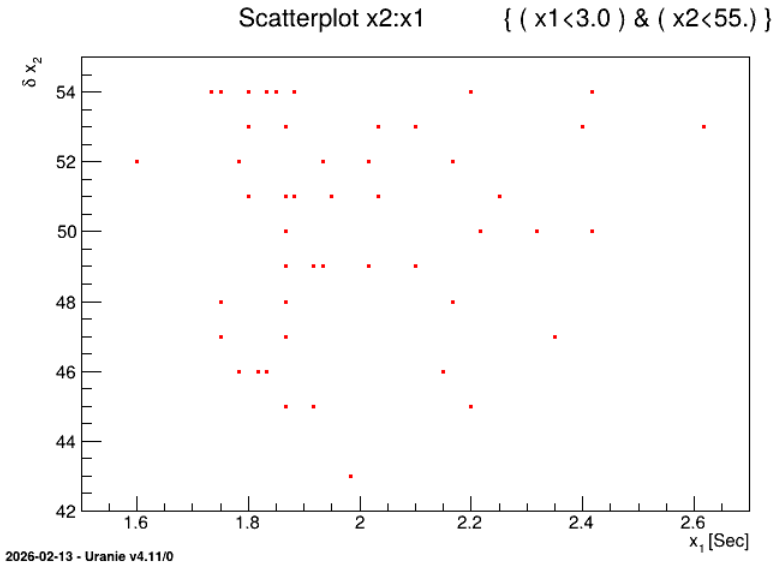


Figure 2.40: Graph with a selection definition

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("x1 >= 3.")
tdsGeyser.draw("x2:x1")
```

The obtained figure is as follows:

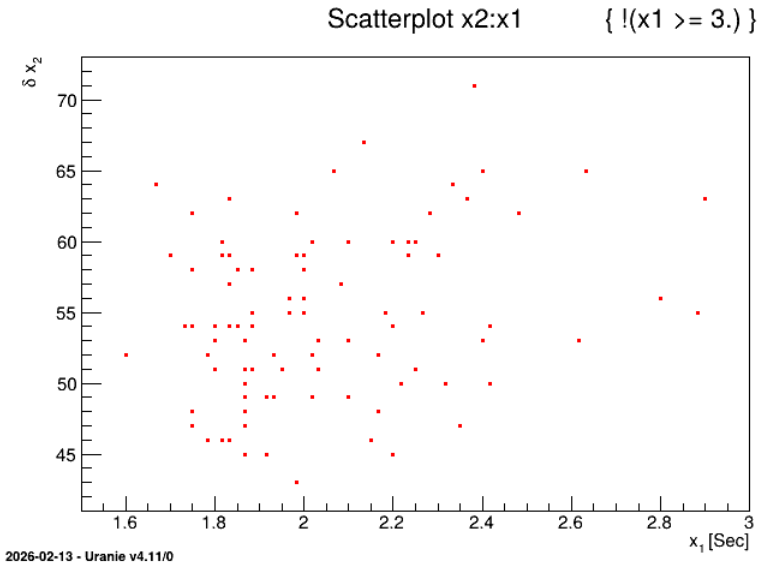


Figure 2.41: Graph with a definition of *Cut*

It can be noticed in the title of Figure 2.41, 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 *List of variable information* 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.

2.3.7 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.

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

tdsGeyser.exportData("newfile.dat") # (1)
tdsGeyser.exportDataHeader("newfile.C", "x1:x2:y") # (2)
tdsGeyser.exportDataNeMo("newfile.nemo", "x1:x2", "y", "x2<75.0") # (3)
tdsGeyser.exportDataJSon("newfile.json") # (4)
```

Data export from a `TDataServer` in an ASCII file

1. 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
```

2. 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,
```

(continues on next page)

(continued from previous page)

```
...
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
```

3. 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
```

4. 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" :
  [

    {
```

(continues on next page)

(continued from previous page)

```
      "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.* *Pattern selection*) 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.* *Pattern selection*) 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.* *Pattern selection*) of the `TDataServer`.

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

  Exportation of the `TDataServer` data in 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.* *Pattern selection*) of the `TDataServer`

2.4 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 *Normalising the variable*
- The ranking of variable, in *Computing the ranking*
- The elementary statistic computation, in *Computing the elementary statistic*
- The quantile estimation, in *The quantile computation*
- The correlation matrix determination, in *Correlation matrix*

2.4.1 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.

```
"""
Example of vector normalisation
"""
from URANIE import DataServer

tdsop = DataServer.TDataServer("foo", "pouet")
tdsop.fileDataRead("tdstest.dat")

# Compute a global normalisation of v, CenterReduced
tdsop.normalize("v", "GCR", DataServer.TDataServer.kCR, True)
# Compute a normalisation of v, CenterReduced (not global but entry by entry)
tdsop.normalize("v", "CR", DataServer.TDataServer.kCR, False)

# Compute a global normalisation of v, Centered
tdsop.normalize("v", "GCent", DataServer.TDataServer.kCentered)
# Compute a normalisation of v, Centered  (not global but entry by entry)
tdsop.normalize("v", "Cent", DataServer.TDataServer.kCentered, False)

# Compute a global normalisation of v, ZeroOne
tdsop.normalize("v", "GZO", DataServer.TDataServer.kZeroOne)
# Compute a normalisation of v, ZeroOne (not global but entry by entry)
tdsop.normalize("v", "ZO", DataServer.TDataServer.kZeroOne, False)
```

(continues on next page)

(continued from previous page)

```
# Compute a global normalisation of v, MinusOneOne
tdsop.normalize("v", "GMOO", DataServer.TDataServer.kMinusOneOne, True)
# Compute a normalisation of v, MinusOneOne (not global but entry by entry)
tdsop.normalize("v", "MOO", DataServer.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

2.4.2 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:

```
"""
Example of rank usage for illustration purpose
"""
from URANIE import DataServer

tdsGeyser = DataServer.TDataServer("geyser", "poet")
tdsGeyser.fileDataRead("geyser.dat")
tdsGeyser.computeRank("x1")
tdsGeyser.computeStatistic("Rk_x1")

print("NPatterns="+str(tdsGeyser.getNPatterns())+";  min(Rk_x1)= " +
      str(tdsGeyser.getAttribute("Rk_x1").getMinimum())+";  max(Rk_x1)= " +
      str(tdsGeyser.getAttribute("Rk_x1").getMaximum()))
```

This macro should returns

```
NPatterns=272;  min(Rk_x1)= 1.0;  max(Rk_x1)= 272.0
```

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.

2.4.3 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).

```
"""
Example of statistical usage
"""
from URANIE import DataServer
```

(continues on next page)

(continued from previous page)

```
tdsGeyser = DataServer.TDataServer("geyser", "poet")
tdsGeyser.fileDataRead("geyser.dat")
tdsGeyser.computeStatistic("x1")

print("min(x1)= "+str(tdsGeyser.getAttribute("x1").getMinimum())+";  max(x1)= "
      + str(tdsGeyser.getAttribute("x1").getMaximum())+";  mean(x1)= " +
      str(tdsGeyser.getAttribute("x1").getMean())+";  std(x1)= " +
      str(tdsGeyser.getAttribute("x1").getStd()))
```

It returns the following line

```
min(x1)= 1.6;  max(x1)= 5.1;  mean(x1)= 3.487783088235295;  std(x1)= 1.141371251105209
```

2.4.3.1 Specific case of vectors

As stated in *List of variable information*, 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 *Adding attributes to a TDataServer*:

```
"""
Example of statistical estimation on vector attributes
"""
from URANIE import DataServer

tdsop = DataServer.TDataServer("foo", "poet")
tdsop.fileDataRead("tdstest.dat")

# Considering every element of a vector independent from the others
tdsop.computeStatistic("x")
px = tdsop.getAttribute("x")

print("min(x[0])= "+str(px.getMinimum(0))+";  max(x[0])= "+str(px.getMaximum(0))
      + ";  mean(x[0])= "+str(px.getMean(0))+";  std(x[0])= "+str(px.getStd(0)))
print("min(x[1])= "+str(px.getMinimum(1))+";  max(x[1])= "+str(px.getMaximum(1))
      + ";  mean(x[1])= "+str(px.getMean(1))+";  std(x[1])= "+str(px.getStd(1)))
print("min(x[2])= "+str(px.getMinimum(2))+";  max(x[2])= "+str(px.getMaximum(2))
      + ";  mean(x[2])= "+str(px.getMean(2))+";  std(x[2])= "+str(px.getStd(2)))
print("min(xtot)= "+str(px.getMinimum(3))+";  max(xtot)= "+str(px.getMaximum(3))
      + ";  mean(xtot)= "+str(px.getMean(3))+";  std(xtot)= "+str(px.getStd(3)))

# 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:::6")
```

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])= 1.0;  max(x[0])= 7.0;  mean(x[0])= 3.0;  std(x[0])= 3.4641016151377544
min(x[1])= 2.0;  max(x[1])= 8.0;  mean(x[1])= 4.666666666666666;  std(x[1])= 3.
↪055050463303893
min(x[2])= 3.0;  max(x[2])= 9.0;  mean(x[2])= 6.666666666666666;  std(x[2])= 3.
↪2145502536643185
min(xtot)= 1.0;  max(xtot)= 9.0;  mean(xtot)= 4.777777777777778;  std(xtot)= 3.
↪2317865716108862
```

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 *Adding attributes to a TDataServer*. 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(intiel=0)`, `getStd(intiel=0)`, `getMinimum(intiel=0)`, `getMaximum(intiel=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.

2.4.4 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 2.44 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.

2.4.4.1 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.

```
from ctypes import c_double
import numpy as np

aproba = c_double(0.5)
aquant = c_double(0)
tdsGeyser.computeQuantile("x2", aproba, aquant) # (1)

Proba = np.array([0.05,0.95], dtype=np.float64)
Quant = np.array([0.0,0.0], dtype=np.float64) # (2)
tdsGeyser.computeQuantile("x2", 2, Proba, Quant)

print("Quant[0]=%f; Quant[1]=%f" % (Quant[0],Quant[1])) # (3)
```

Description of the methods and results

1. 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.
2. 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.
3. 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 *Computing the elementary statistic*). 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 *Macro "dataserverComputeQuantileVec.py"* shows the different methods created in the attribute class in order for the user to get back the computed values:

```
"""
Example of quantile estimation for many values at once
"""
from sys import stdout
from ctypes import c_int, c_double
import numpy as np
from URANIE import DataServer
import ROOT

tdsvec = DataServer.TDataServer("foo", "bar")
tdsvec.fileDataRead("aTDSWithVectors.dat")

probas = np.array([0.2, 0.6, 0.8], 'd')
quants = np.array(len(probas)*[0.0], 'd')
tdsvec.computeQuantile("rank", len(probas), probas, quants)

prank = tdsvec.getAttribute("rank")
nbquant = c_int(0)
prank.getQuantilesSize(nbquant)  # (1)
print("nbquant = " + str(nbquant.value))

aproba = c_double(0.8)
aquant = c_double(0)
prank.getQuantile(aproba, aquant)  # (2)
print("aproba = " + str(aproba.value) + ", aquant = " + str(aquant.value))

theproba = np.array(nbquant.value*[0.0], 'd')
thequant = np.array(nbquant.value*[0.0], 'd')
prank.getQuantiles(theproba, thequant)  # (3)
for i_q in range(nbquant.value):
    print("(theproba, thequant)[" + str(i_q) + "] = (" + str(theproba[i_q]) +
          ", " + str(thequant[i_q]) + ")")

allquant = ROOT.vector('double')()
prank.getQuantileVector(aproba, allquant)  # (4)
stdout.write("aproba = " + str(aproba.value) + ", allquant = ")
for quant_i in allquant:
```

(continues on next page)

(continued from previous page)

```
    stdout.write(str(quant_i) + " ")
print("")
```

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.

2.4.4.2 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.* *The Modeler Module*). This model will enable the creation of the control variate.

```
# Build the SR ( Linear regression + ANN)
tlin = Modeler.TLinearRegression(tds, "rw:r:tu:tl:hu:hl:l:kw", sY, "Dummy")
tlin.estimate()
tlin.exportFunction("c++", "_SR_rl_", "SRrl")

tann = Modeler.TANNModeler(tds, "%s,8,%s" % (sinput,sY) )
tann.train(3, 2, "test")
tann.setDrawProgressBar(False)
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
ROOT.gROOT.LoadMacro("_SR_rl_.C")
tlfz = Launcher.TLauncherFunction(tds, "SRrl", sinput, "Z")
tlfz.setDrawProgressBar(False)
tlfz.run()
```

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

```
tdsza = DataServer.TDataServer( "%s_zalpha" %(tds2.GetName()), "Ex. flowrate")
for i in range(nattinput):
    tdsza.addAttribute( tds2.getAttribute(i))

fsza = Sampler.TSampling(tdsza, "lhs", 6000)
fsza.generateSample()

tlfrlza = Launcher.TLauncherFunction(tdsza, "SRrl", sinput, "Zrl")
tlfrlza.setDrawProgressBar(False)
tlfrlza.run()

from ctypes import c_double
dAlpha = c_double(0.5)
dZrla = c_double(0)
tdsza.computeQuantile("Zrl", dAlpha, dZrla)
print(dZrla)
```

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

```
dY = c_double(0)
rho = c_double(0)
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).

```
tis = Sampler.TImportanceSampling(tds2, "rw", DataServer.TNormalDistribution("rw_IS",␣
↪0.10, 0.015), nS)
```

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

```
tdsis = tis.getTDS()
```

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

```
sampis = Sampler.TSampling(tdsis,"lhs",nS)
sampis.generateSample()
tlfis = Launcher.TLauncherFunction(tdsis, "flowrateModel", "*", "Y_IS")
tlfis.setDrawProgressBar(False)
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.

```
ISmean = tis.computeMean("Y_IS")
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.

2.4.4.3 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 *computeQuantile*: 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 *computeQuantile*). 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 [Bla17]). 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:

```
tds = DataServer.TDataServer("useless","foo")
quant=0.95
CL = 0.95
SampleSize=200
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(DataServer.TNormalDistribution("attribute",0,1))
sampis = Sampler.TSampling(tds,"lhs", SampleSize)
sampis.generateSample()

from ctypes import c_double
value=c_double(0)
tds.estimateQuantile("attribute", quant, value, CL)
```

As stated previously, this is illustrated in a use-case macro which results in Figure 13.6. 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.

2.4.5 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.

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

matCorr = tdsGeyser.computeCorrelationMatrix("x2:x1")
print("Computing correlation matrix ...")
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:

```
matCorrRank = tdsGeyser.computeCorrelationMatrix("x2:x1", "", "rank")
print("Computing correlation matrix on ranks ...")
matCorrRank.Print()
```

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

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

2.4.5.1 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 *Adding attributes to a TDataServer*, 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.

```
"""
Example of correlation matrix computation for vector
"""
from URANIE import DataServer
```

(continues on next page)

(continued from previous page)

```
tdsop = DataServer.TDataServer("foo", "poet")
tdsop.fileDataRead("tdstest.dat")

# Consider a and x attributes (every element of the vector)
globalOne = tdsop.computeCorrelationMatrix("x:a")
globalOne.Print()

# Consider a and x attributes (cherry-picking a single element of the vector)
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 *Macro "dataserverComputeCorrelationMatrixVector.py"*.

```
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**).

2.5 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.

2.5.1 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.

```
c2 = ROOT.TCanvas()
apad = ROOT.TPad("apad","apad",0, 0.03, 1, 1)

tdsGeyser = DataServer.TDataServer("tdsgeyser", "Database of the geyser")
tdsGeyser.fileDataRead("geyser.dat")
tdsGeyser.addAttribute("xnorm", "sqrt(x1*x1+x2*x2)")

apad.Divide(2,2)

apad.cd(1)
tdsGeyser.draw("xnorm","","nclass=root")
apad.cd(2)
tdsGeyser.draw("xnorm","","nclass=sturges")
apad.cd(3)
tdsGeyser.draw("xnorm","","nclass=fd")
apad.cd(4)
tdsGeyser.draw("xnorm","","nclass=scott")
```

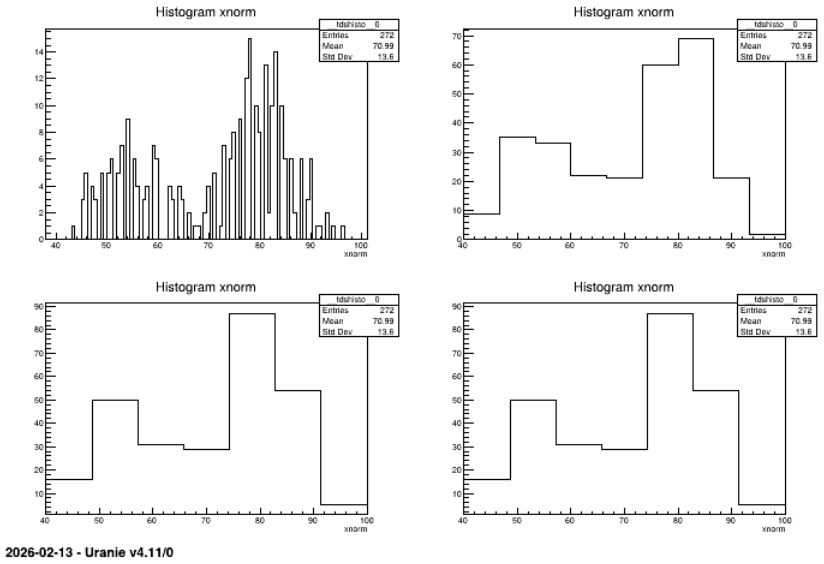


Figure 2.42: Different histograms of the same attribute *xnorm* depending on the method for computing bins. The values are respectively 100(root), 8 from sturges, 7 from fd and scoot.

2.5.2 Box-and-whisker("boxplot")

```
tdsGeyser.drawBoxPlot("x2")
```

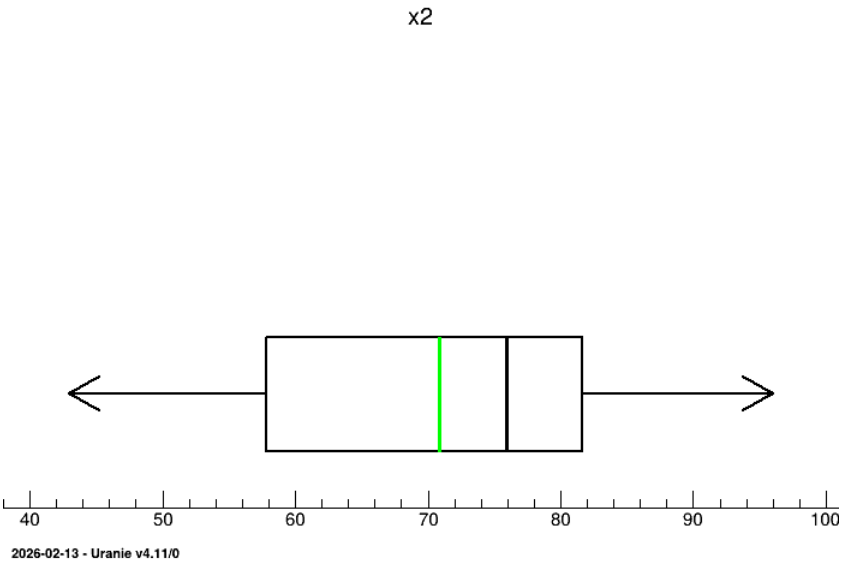


Figure 2.43: Boxplot of attribute *x2* of the `TDataServer geyser`

2.5.3 CDF, CCDF curves

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

```
tdsGeyser.drawCDF("x2")
```

```
tdsGeyser.drawCDF("x2","","ccdf")
```

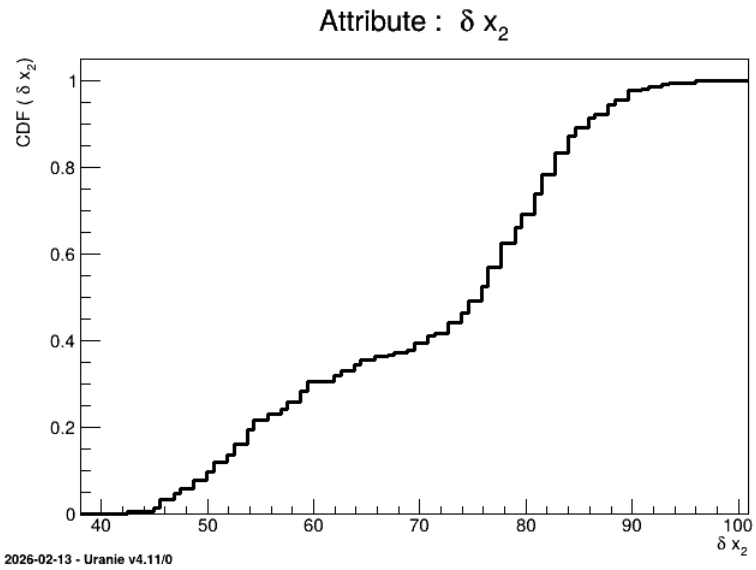


Figure 2.44: CDF graph of attribute *x2* of the `TDataServer geyser`

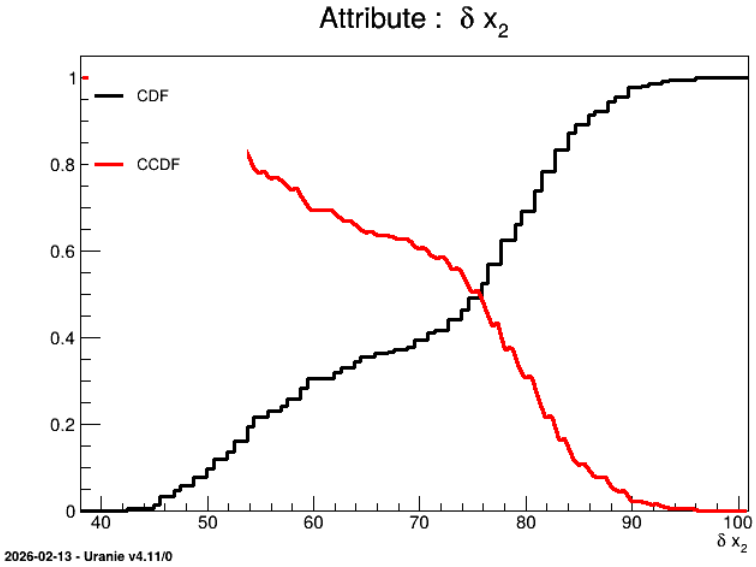


Figure 2.45: Graphs CDF+CCDF of the attribute *x2* of the `TDataServer geyser`

2.5.4 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.

```
tdsGeyser = DataServer.TDataServer("tdsgeyser", "Geyser database")
tdsGeyser.fileDataRead("geyser.dat")
```

(continues on next page)

(continued from previous page)

```
tdsGeyser.drawScatterplot("x2:x1")
```

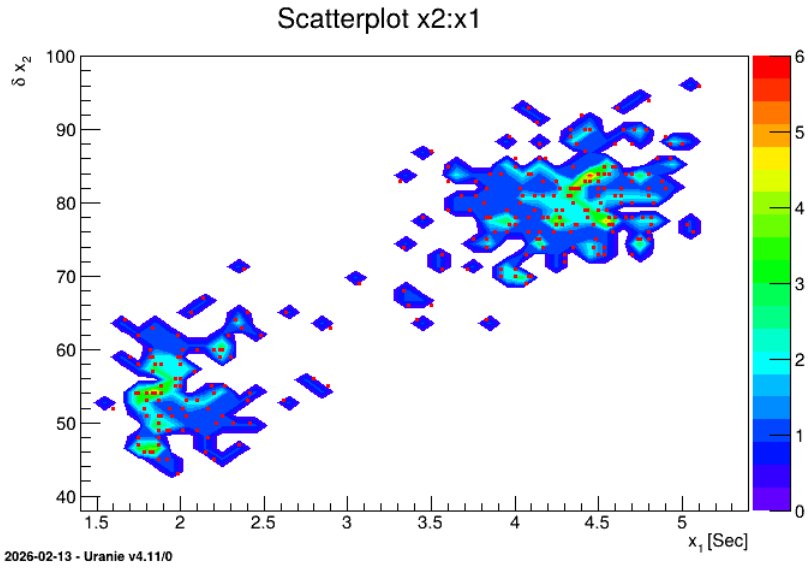


Figure 2.46: Scatterplot between attributes x1 and x2 of the `TDataServer geyser`.

This figure has to be compared to the classical scatterplot shown, for instance, in Figure 2.4.

2.5.5 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 2.47).

```
tdsGeyser = DataServer.TDataServer("tdsgeyser", "Geyser database")
tdsGeyser.fileDataRead("geyser.dat")
tdsGeyser.drawProfile("x2:x1","","same")
```

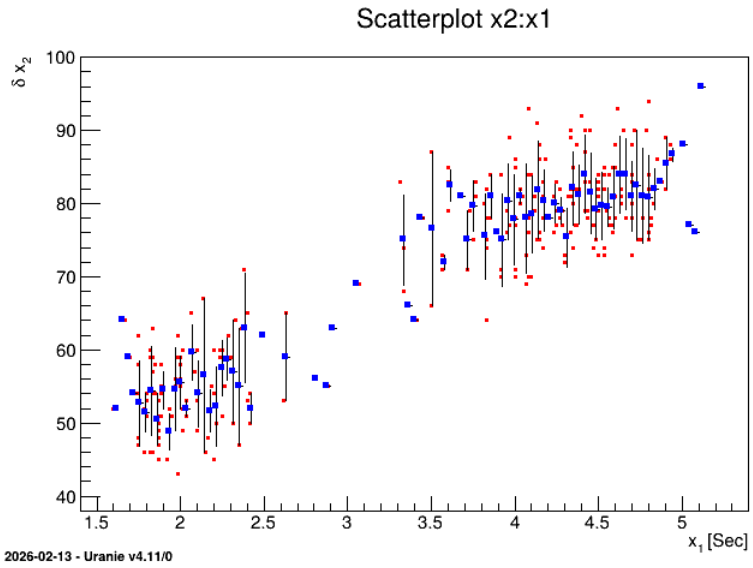


Figure 2.47: Scatterplot between attributes x1 and x2 of the `TDataServer geyser`.

2.5.6 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.

```
tdsGeyser = DataServer.TDataServer("tdsgeyser", "Database of the geyser")
tdsGeyser.fileDataRead("geyser.dat")
tdsGeyser.drawTufte("x2:x1")
```

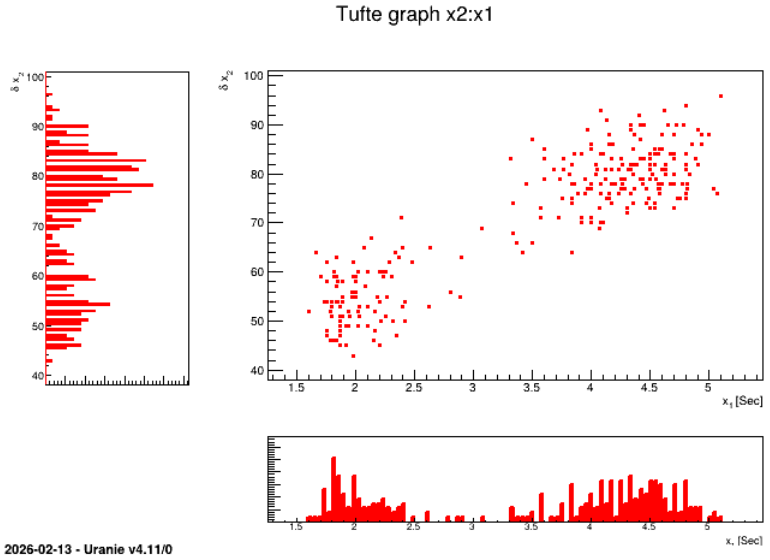


Figure 2.48: Graphs of "Tufte" type between the attributes x1 and x2 of the `TDataServer geyser`.

2.5.7 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.

```
tdsGeyser = DataServer.TDataServer("tdsgeyser", "Database of the geyser")
tdsGeyser.fileDataRead("flowrateUniformDesign.dat")
tdsGeyser.drawPairs()
```

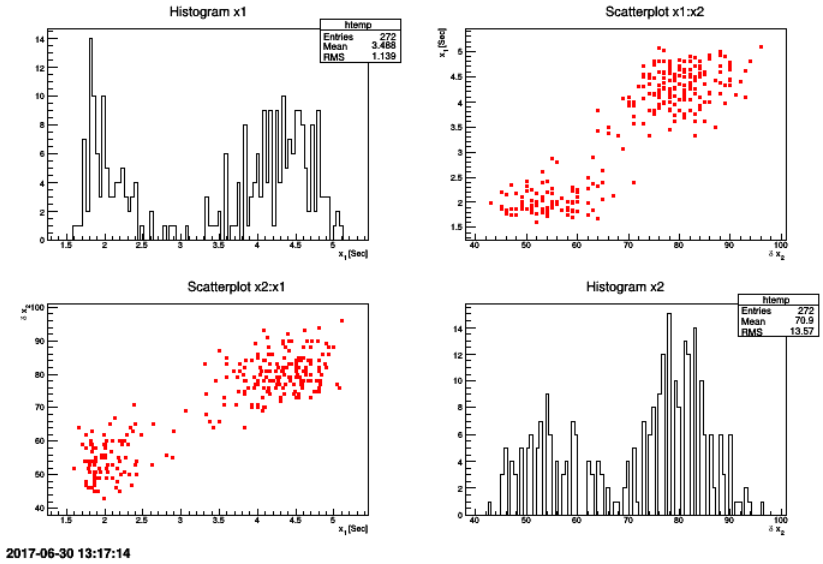


Figure 2.49: Graphs of “Pairs” of the `TDataServer geyser`

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 ":"

2.5.8 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 2.50 (obtained with the code below). No special trend can be seen here.

```
tdsCobweb = DataServer.TDataServer("tdscobweb", "Database of the cobweb")
tdsCobweb.fileDataRead("cobwebdata.dat") # read data file
tdsCobweb.drawPairs() # do the drawPairs graph
```

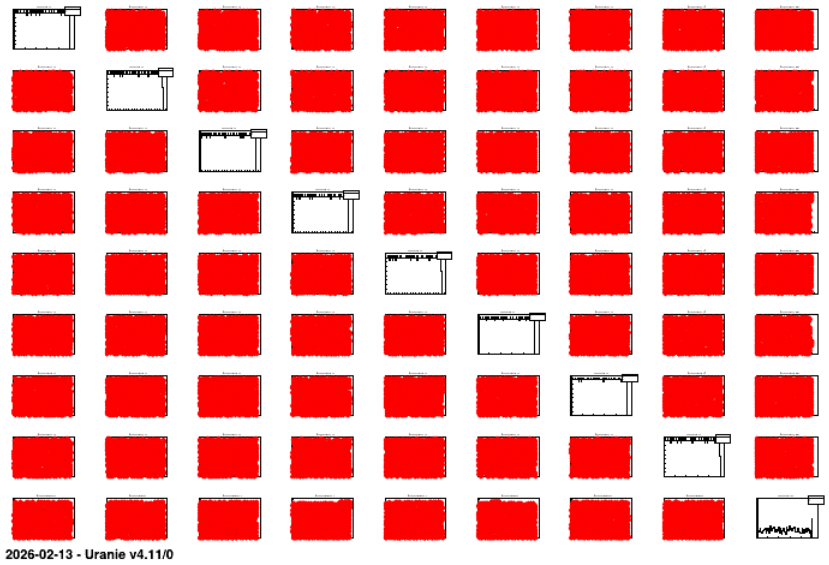


Figure 2.50: Graphs of "drawPairs" type between the 8 uniformly-distributed inputs and the output of a given problem.

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
para = ROOT.gPad.GetListOfPrimitives().FindObject("__tdspara__0")
# Get the output axis
axis = para.GetVarList().FindObject("out")

# Create a range for 0.97 < out < 1.0 and display it in blue
Range  = ROOT.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 2.51.

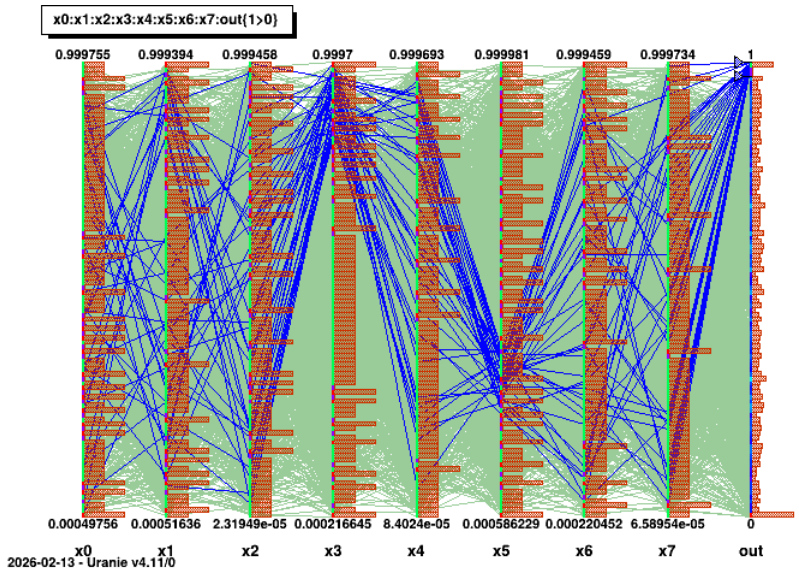


Figure 2.51: Graphs of "CobWeb" type between the 8 uniformly-distributed inputs and the output of a given problem.

2.5.9 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.

```
tdsQQ = DataServer.TDataServer("tdsQQ", "Database of the QQ")
tdsQQ.fileDataRead("geyser.dat") # read data file
tdsQQ.computeStatistic() # to estimate mu and sigma
tdsQQ.drawQQPlot("x2", "normal(%g,%g)" % (tdsQQ.getAttribute("x2").getMean(), tdsQQ.
↪getAttribute("x2").getStd()), 400)
```

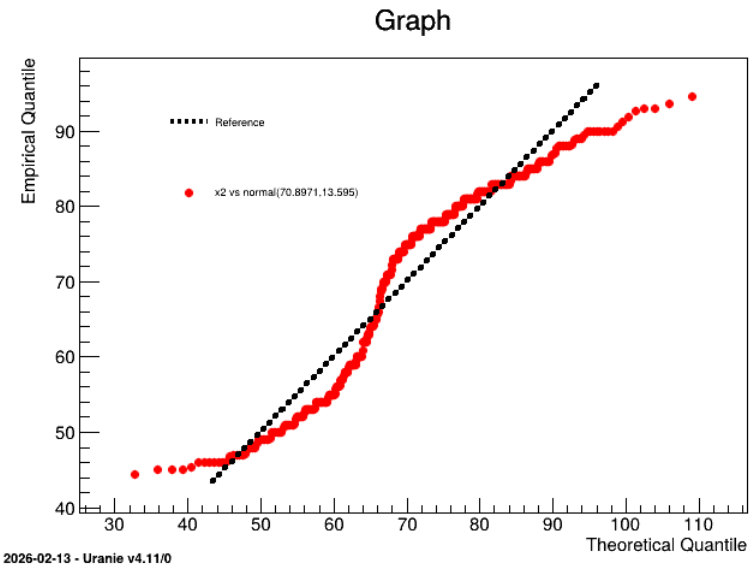


Figure 2.52: Plot resulting from the "drawQQPlot" method, comparing "x2" to a normal distribution.

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 2.48 for instance). An heavy use of this method is provided in *Macro "dataserverDrawQQPlot.py"*.

2.5.10 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.

```
tdsPP = DataServer.TDataServer("tdsPP", "Database of the PP")
tdsPP.fileDataRead("geyser.dat") # read data file
tdsPP.computeStatistic() # to estimate mu and sigma
tdsPP.drawPPPlot("x2", "normal(%g,%g)" %(tdsPP.getAttribute("x2").getMean(), tdsPP.
↪getAttribute("x2").getStd()), 400)
```

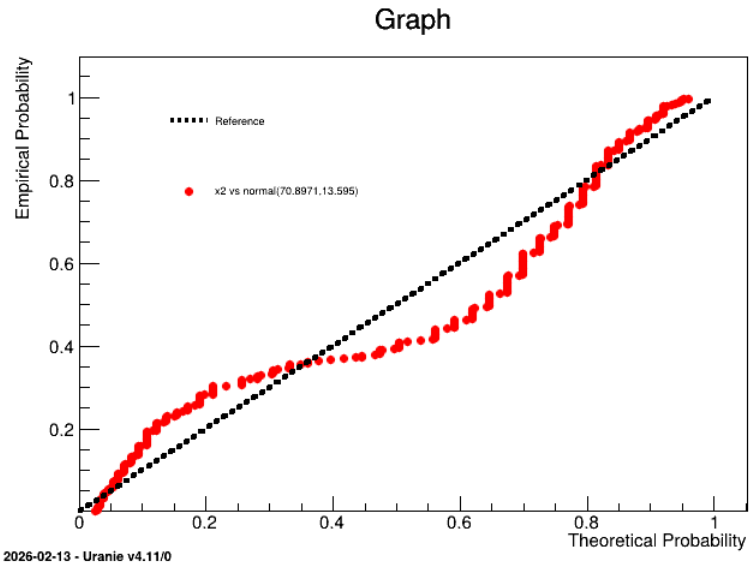


Figure 2.53: Plot resulting from the "drawPPPlot" method, comparing "x2" to a normal distribution.

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 2.48 for instance). An heavy use of this method is provided in *Macro "dataserverDrawPPPlot.py"*.

2.6 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 [Jol11]) a small theoretical introduction can in any case be found in [Bla17].

2.6.1 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
```

(continues on next page)

(continued from previous page)

```
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 2.54, 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.

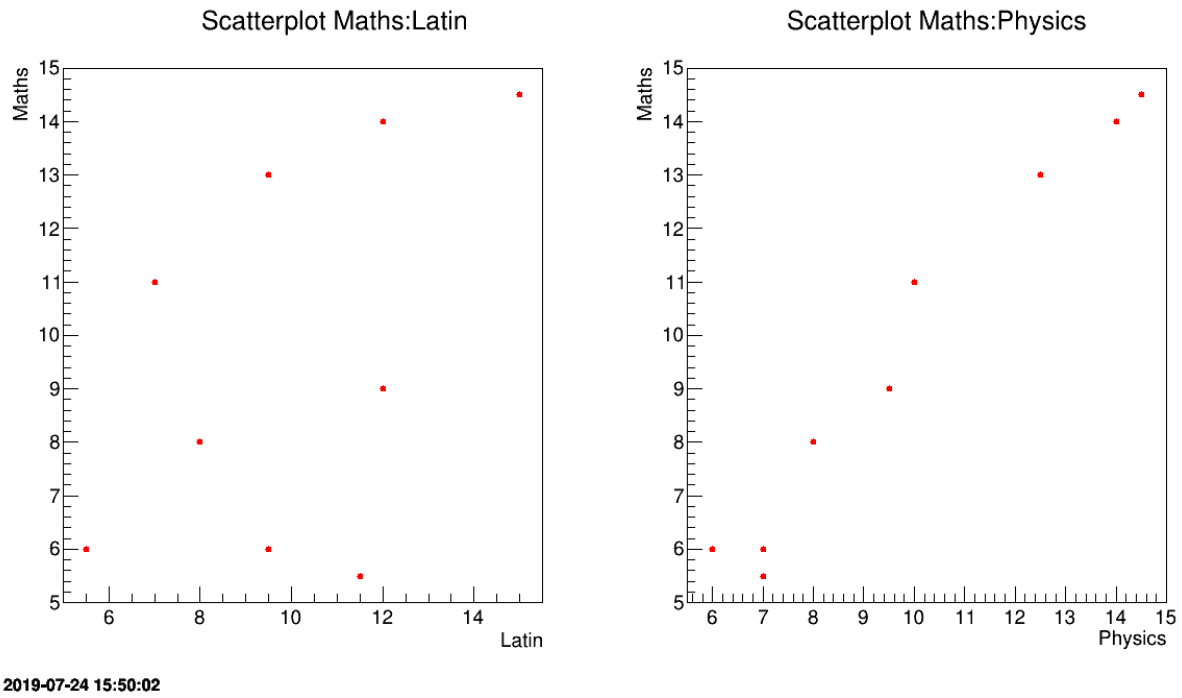


Figure 2.54: Representation of some variables of the Notes sample.

2.6.1.1 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
tdsPCA = DataServer.TDataServer("tdsPCA", "my TDS")
tdsPCA.fileDataRead("Notes.dat")

# Create the PCA object precising the variables of interest
tpca = DataServer.TPCA(tdsPCA, "Maths:Physics:French:Latin:Music")
tpca.compute()
```

Once done, the process described in the [Bla17] is finished and then interpretation starts.

2.6.1.2 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 *Macro "dataserverPCAExample.py"* to see the results).

The following lines are showing how to represent the eigenvalues as plots and the results are gathered in Figure 2.55. 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
c = ROOT.TCanvas("cEigenValues", "Eigen Values Plot", 1100, 500)
apad3 = ROOT.TPad("apad3", "apad3", 0, 0.03, 1, 1)
apad3.Draw()
apad3.cd()
apad3.Divide(3, 1)
ntd = tpca.getResultNtupleD()
apad3.cd(1)
ntd.Draw("eigen:i", "", "lp")
apad3.cd(2)
ntd.Draw("eigen_pct:i", "", "lp")
ROOT.gPad.SetGrid()
apad3.cd(3)
ntd.Draw("sum_eigen_pct:i", "", "lp")
ROOT.gPad.SetGrid()
c.SaveAs("PCA_notes_musique_Eigen.png")
```

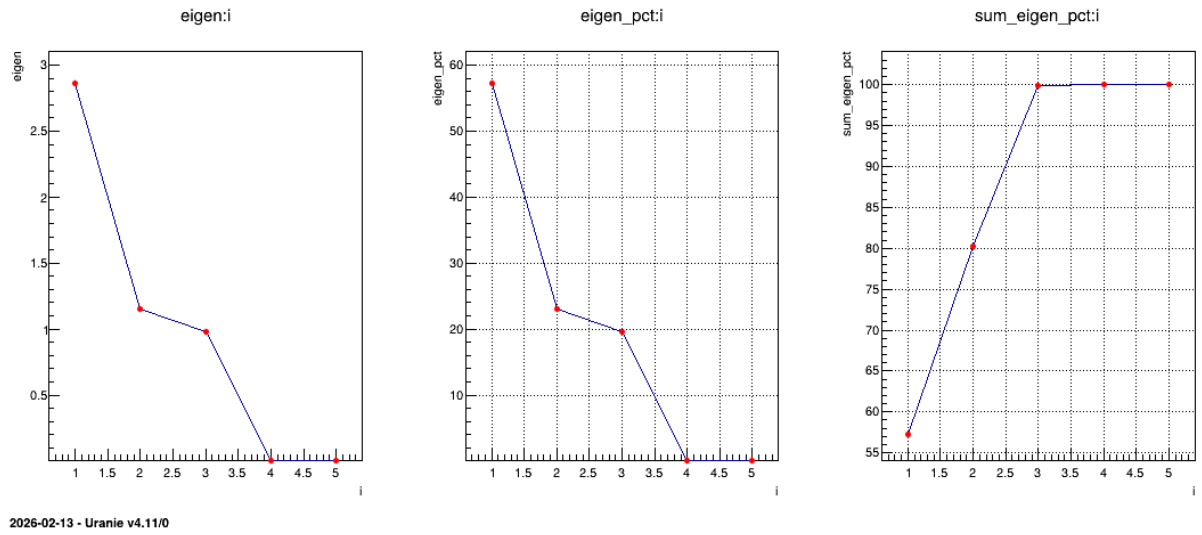


Figure 2.55: Representation of the eigenvalues (left) their overall contributions in percent (middle) and the sum of the contributions (right) from the PCA analysis.

As stated previously, one can have a look at the variables in their usual representation (that can be seen in Figure 2.56) 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 2.56 is shown below:

```
# Draw all variable weight in PC definition
cLoading = ROOT.TCanvas("cLoading", "Loading Plot", 800, 800)
apad2 = ROOT.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 2.56 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 2.56 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 2.56 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.

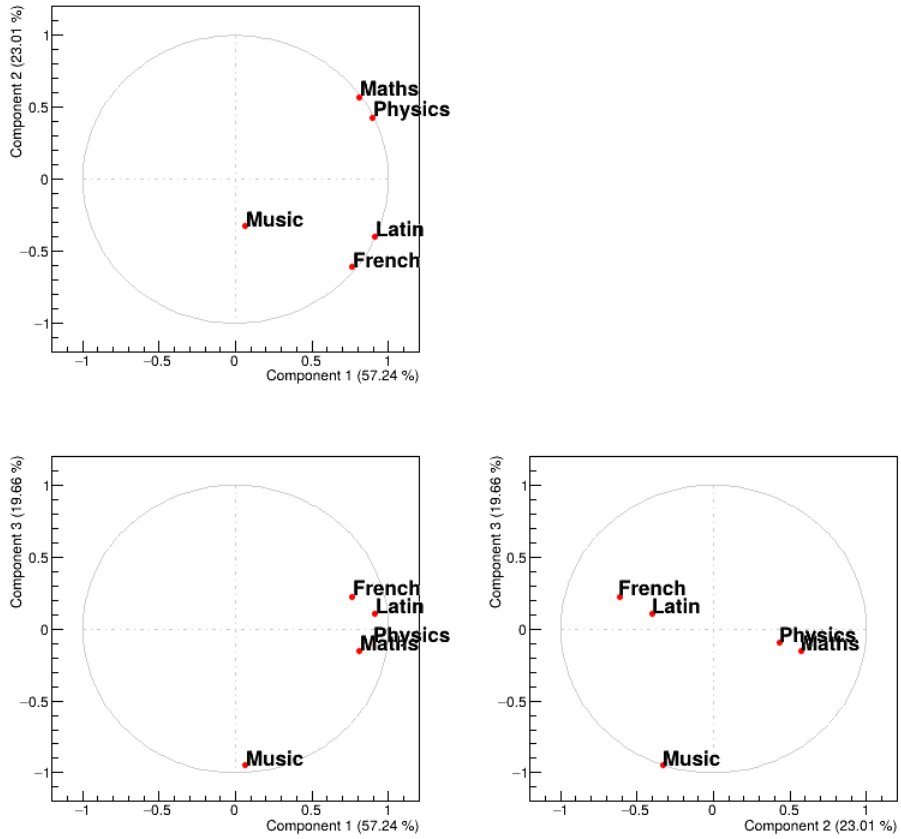


Figure 2.56: Representation of correlation between the original variables and the PC under study.

Finally, one can also have a look at the points distribution in the new PC space (that can be seen in Figure 2.57). 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 2.57 is shown below:

```
# Draw all point in PCA planes
cPCA = ROOT.TCanvas("cpca", "PCA", 800, 800)
apad1 = ROOT.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 2.57 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.

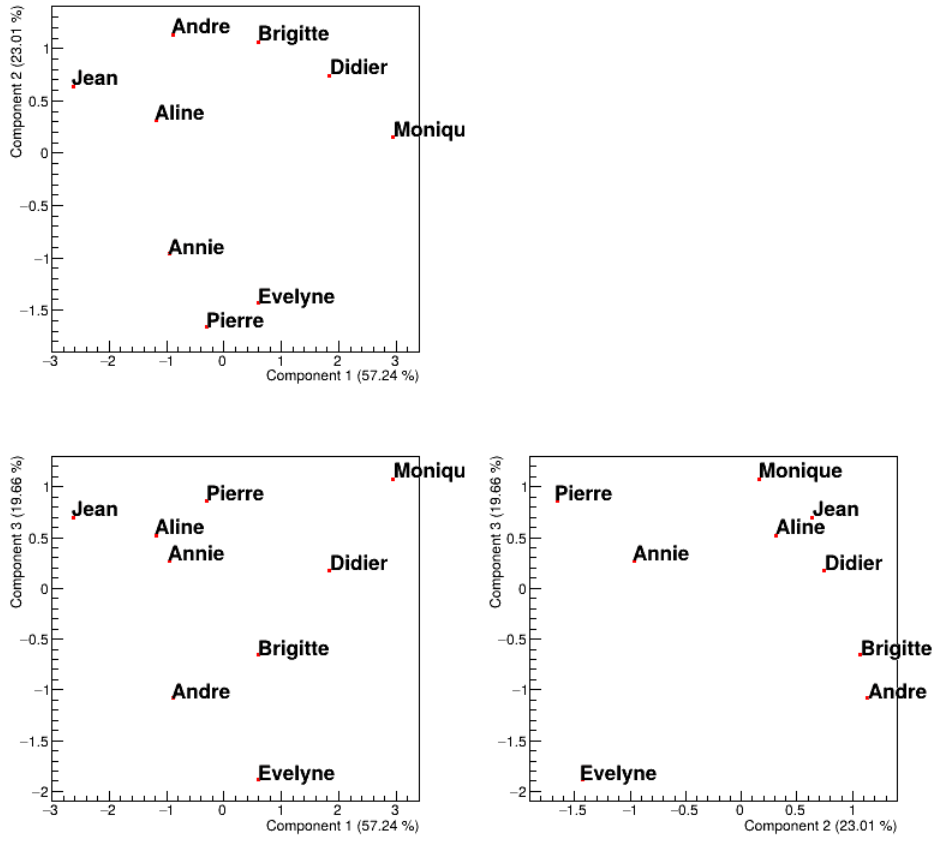


Figure 2.57: Representation of the data points in the PC-defined plane.

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 *Macro "dataserverPCAExample.py"*.

CHAPTER

THREE

THE SAMPLER MODULE

Abstract This is a description of the Sampler module whose main goal is to produce a design-of-experiments, starting from the description of the model provided by the user, and that would be used to investigate the sensitivity and/or propagate uncertainty for the requested analysis.

The source files are in **souRCE.git/meTIER/sampler/souRCE** and the corresponding namespace is **URANIE::Sampler**.

3.1 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:

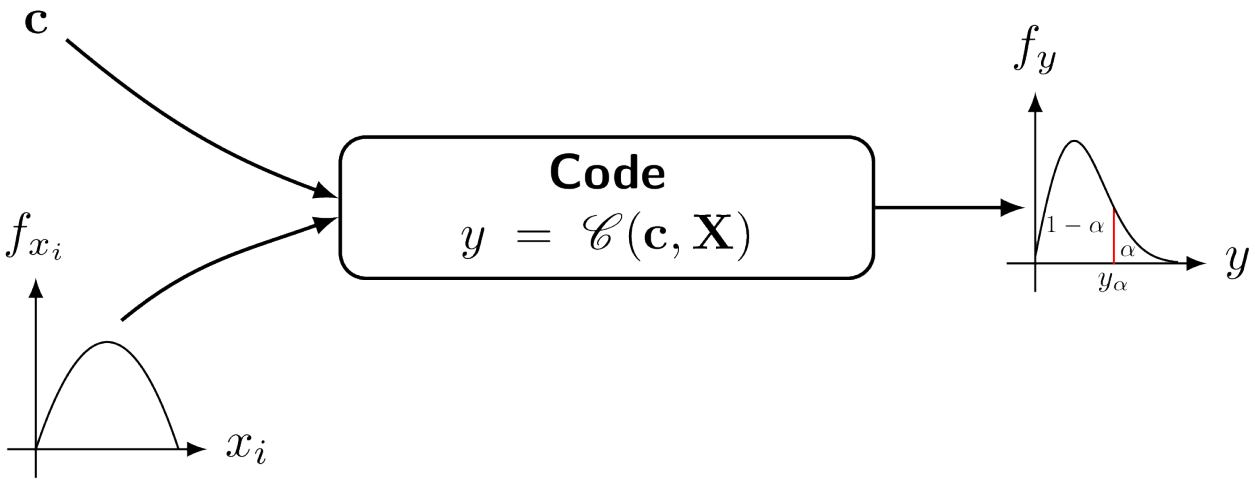


Figure 3.1: Schematic view of the input/output relation through a code

- 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 *The Stochastic methods*). These methods consist in using a random number generator to produce new samples. This is also called Monte-Carlo.
2. **deterministic** methods (see *QMC method*). 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).

3.2 The Stochastic methods

3.2.1 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 *Introducing the TStochasticAttribute classes*

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 3.3-left.

LHS (Latin Hypercube Sampling):
this method [MBC00] 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 3.3-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) [HD02]. 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 3.2, we present two samples of size $N_{\text{calc}} = 8$ 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 3.3 with the same laws, both for SRS on the left and LHS on the right. The "tufte" representation (presented in *Graph 2D "Tufte"*) clearly shows the difference between both methods when considering one-dimensional distribution.

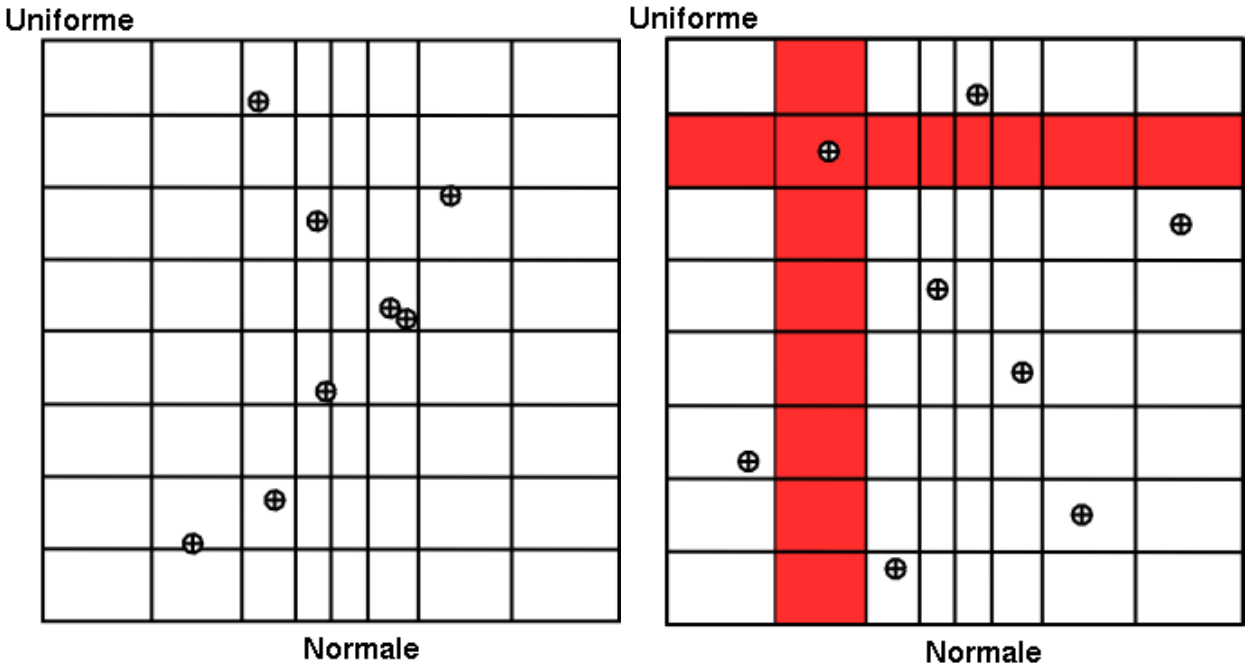


Figure 3.2: Comparison of the two sampling methods SRS (left) and LHS (right) with samples of size 8.

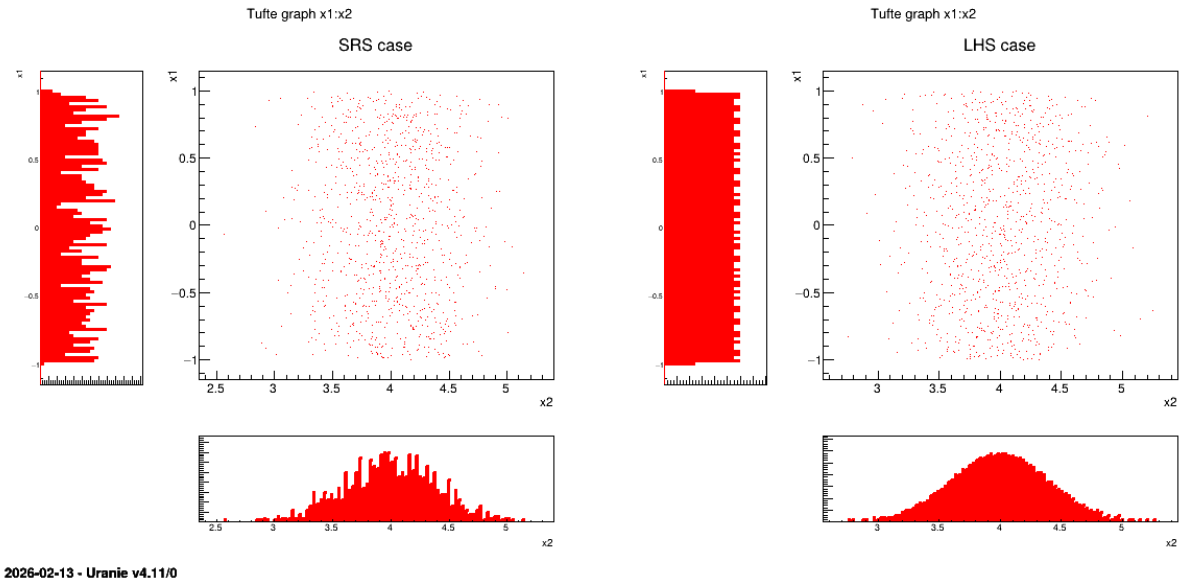


Figure 3.3: Comparison of deterministic design-of-experiments obtained using either SRS (left) or LHS (right) algorithm, when having two independent random variables (uniform and normal one)

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 *The main sampler classes*.
- 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 *The main sampler classes* and in *TConstrLHS example*.

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 *Description of a correlation*

3.2.2 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 ([IC82]) 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 [Bla17] 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 [Bla17]). 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 13.14 which was procuded with the macro introduced in *Macro "samplingMaxiMinLHSFromLHSGrid.py"*;

  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

  ```
  mindist = Sampler.TMaxiMinLHS.getMinDist(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 *TConstrLHS example*. The result of this procedure can be seen in Figure 13.15 which was procuded with the macro introduced in *Macro "samplingConstrLHSLinear.py"*;
- **`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 [Bla17]) 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.

3.2.3 Simple example

The following piece shows how to generate a sample using the `TSampling` class.

```
xunif = DataServer.TUniformDistribution("x1", 3., 4.)  # 1
xnorm = DataServer.TNormalDistribution("x2", 0.5, 1.5)  # 2

tds = DataServer.TDataServer("tdsSampling", "Demonstration Sampling")
tds.addAttribute(xunif)
tds.addAttribute(xnorm)

# Generate the sampling from the TDataServer object
sampling = Sampler.TSampling(tds, "lhs", 1000)  # 3
sampling.generateSample()  # 4
tds.drawTufte("x1:x2")  # 5
```

Generation of a design-of-experiments with stochastic attributes

1. Generating a uniform random variable in [3., 4.]
2. Generating a gaussian random variable with a mean of 0.5 and a standard deviation of 1.5
3. Construct the sampler object requesting a sample of 1000 events, using the Lhs algorithm.
4. Method to generate the sample.
5. Draw the plot as the right side of Figure 3.3

3.2.4 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 [Bla17]. 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 *Macro "samplingConstrLHSLinear.py"*.

```
void Linear(double *p, double *y)
{
    double p1=p[0], p2=p[1];
    double p3=p[2], p4=p[3];

    // Linear constaint
```

(continues on next page)

(continued from previous page)

```
    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 [Bla17] 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 [Bla17] 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:

```
inputs = numpy.array([1, 0, 2, 1], dtype='i')
constrlhs.addConstraint(ROOT.Linear, 2, len(inputs), inputs)
```

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

$$x_1 - x_0 < 0 \quad \Leftrightarrow \quad x_0 - x_1 > 0$$

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).

3.3 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.

3.3.1 Imposing the correlation coefficients

It is possible to set the correlation coefficient by hand, using the `setUserCorrelation` method of both `TSampling` and `TBasicSampling`.

```
xunif = DataServer.TUniformDistribution("x1", 3., 4.)  # 1
xnorm = DataServer.TNormalDistribution("x2", 0.5, 1.5)  # 2

tds = DataServer.TDataServer("tdsSampling", "Demonstration Sampling")
tds.addAttribute(xunif)
tds.addAttribute(xnorm)

# Generate the sampling from the TDataServer object
sampling = Sampler.TSampling(tds, "lhs", 1000)
sampling.setUserCorrelation(xunif, xnorm, 0.90)  # 3
sampling.generateSample()  # 4
tds.Draw("x2:x1", "", "")
```

Specification of a correlation

1. Generating a uniform random variable in [3., 4.]
2. Generating a gaussian random variable with 0.5 as mean value 1.5 as standard deviation.
3. Specification of correlation of 0.90 between the two attributes `xunif` and `xnorm`.
4. Generate the sample by calling the `generateSample` method in `TSampling` (respectively the `generateCorrSample` method in `TBasicSampling`).

The coefficient provided is changing the correlation between the two considered attributes (for a more general discussion on this, see [Bla17]). As an example, Figure 3.4 and Figure 3.5 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 [Bla17]).

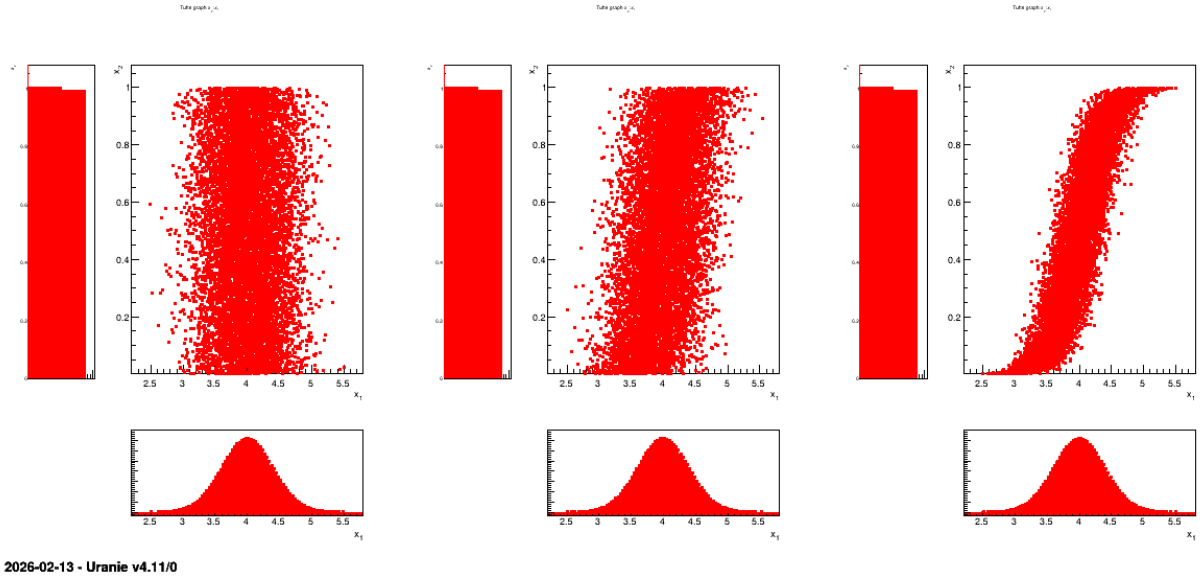


Figure 3.4: Tufte plot of the design-of-experiments created using a normal and uniform distribution, with a LHS method with three correlation coefficient: 0, 0.45 and 0.9

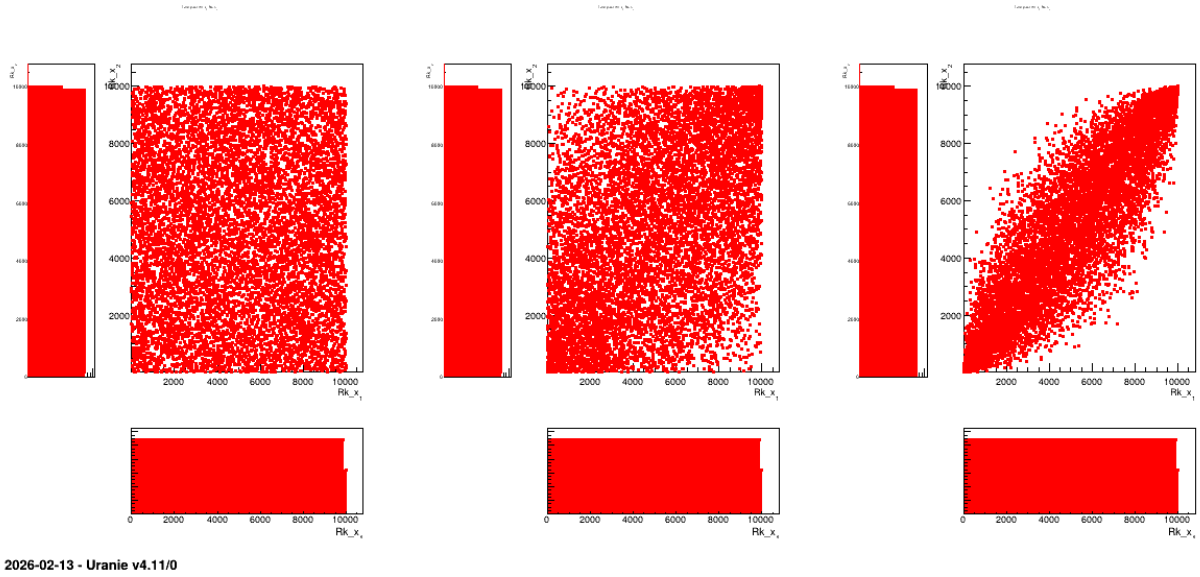


Figure 3.5: Tufte plot of the rank of the design-of-experiments created using a normal and uniform distribution, with a LHS method with three correlation coefficient: 0, 0.45 and 0.9

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).

3.3.1.1 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 [Bla17]. 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:

```
tbs = Sampler.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:

```
ts = Sampler.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 *Macro "samplingSingularCorrelationCase.py"*.

3.3.2 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 3.6 to Figure 3.9 for different θ values along with the formula of the corresponding parametric function.

- **TElliptical:** it is a class to interface the `TGaussian` copula method.

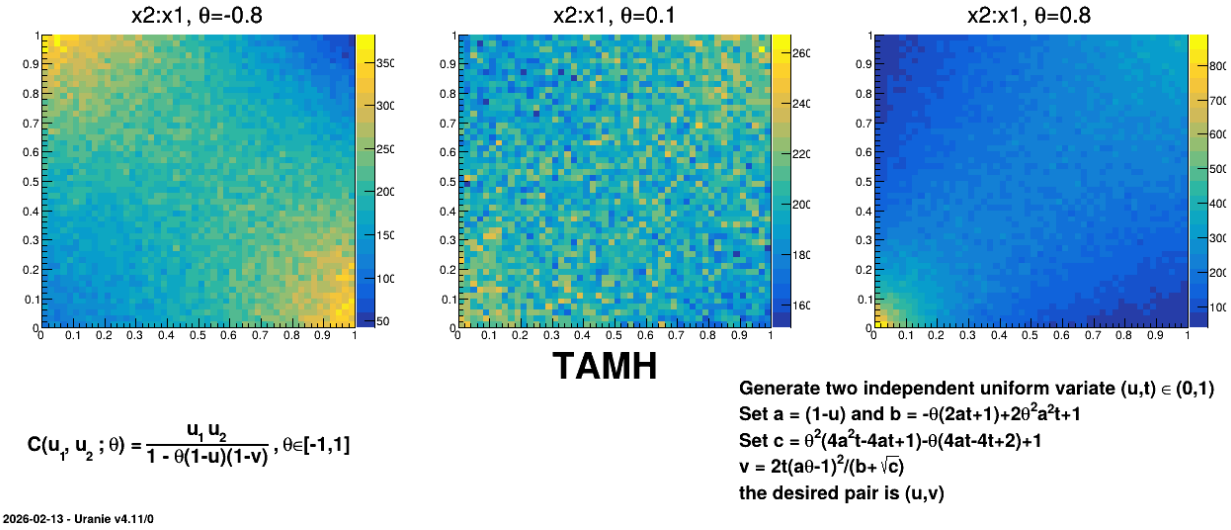


Figure 3.6: Example of sampling done with half million points and two uniform attributes (from 0 to 1), using AMH copula and varying the parameter value.

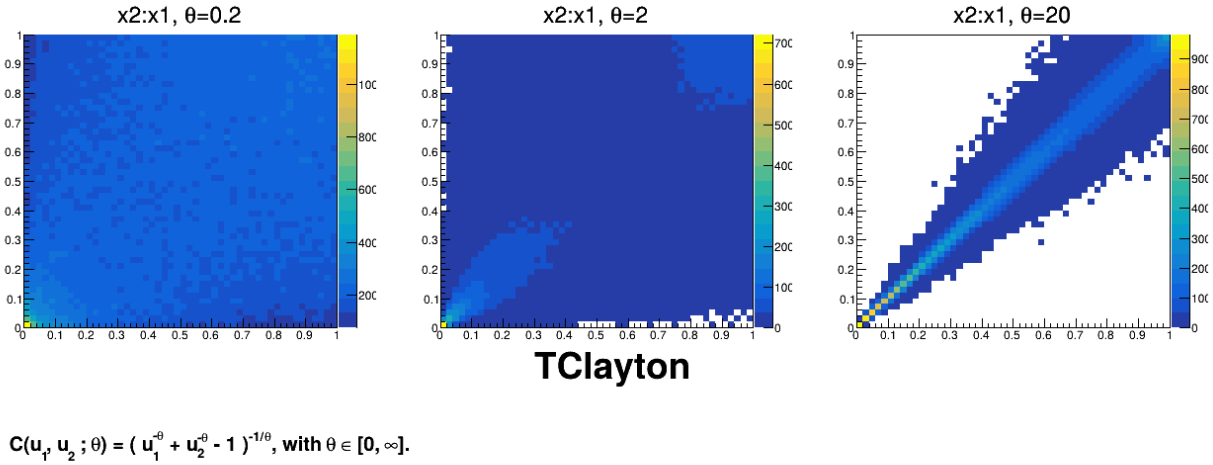


Figure 3.7: Example of sampling done with half million points and two uniform attributes (from 0 to 1), using Clayton copula and varying the parameter value.

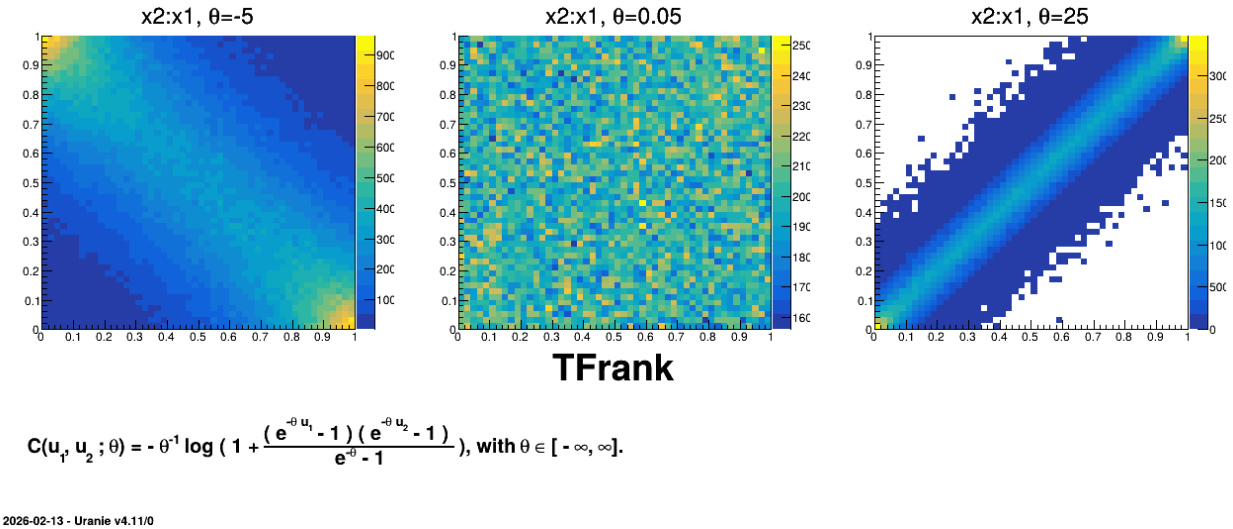


Figure 3.8: Example of sampling done with half million points and two uniform attributes (from 0 to 1), using Frank copula and varying the parameter value.

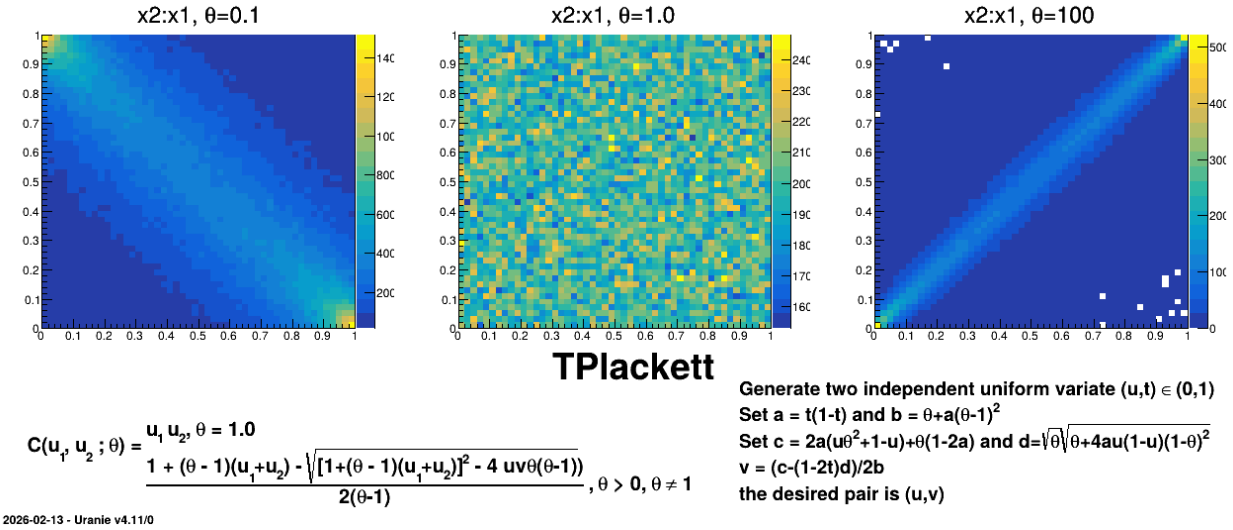


Figure 3.9: Example of sampling done with half million points and two uniform attributes (from 0 to 1), using Plackett copula and varying the parameter value.

The following example shows how to create a copula and to use it in order to get a correlated sample.

```
xunif = DataServer.TUniformDistribution("x1", 3., 4.)
xnorm = DataServer.TNormalDistribution("x2", 0.5, 1.5)

tds = DataServer.TDataServer("tdsSampling", "Demonstration Sampling")
tds.addAttribute(xunif)
tds.addAttribute(xnorm)
```

(continues on next page)

(continued from previous page)

```
# Generate the sampling from the TDataServer object
tamh = Sampler.TAMHCopula(tds, 0.75, 1000)
tamh.generateSample()
tds.Draw("x2:x1", "", "colZ")
```

3.4 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** [Sobol67]
2. Sequences of **Halton** [Hal64]

Figure 3.10 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.

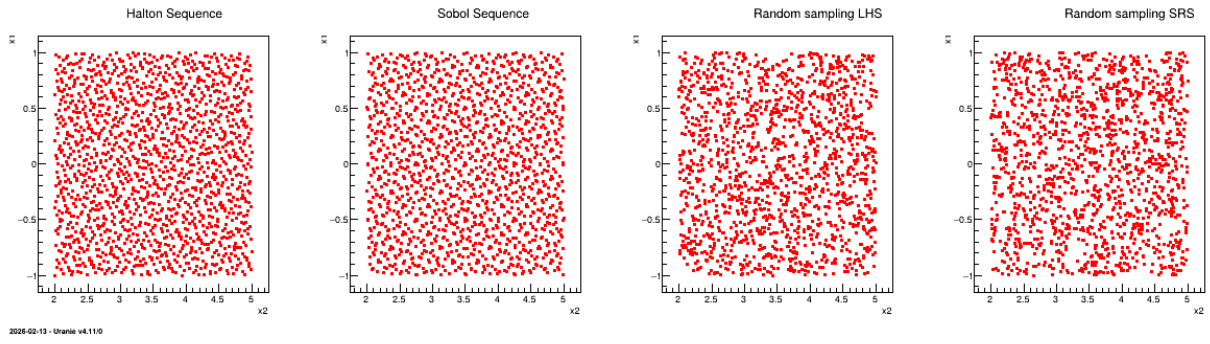


Figure 3.10: Comparison of both quasi Monte-Carlo sequences with both LHS and SRS sampling when dealing with two uniform attributes.

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 *The integration method*. In Uranie we can used the Petras algorithm [Pet01] to produce these sparse grids, shown for different levels in Figure 3.11, that can be compared to regular algorithms ones in Figure 3.10 (in both cases, the problem is described with two uniform attributes).

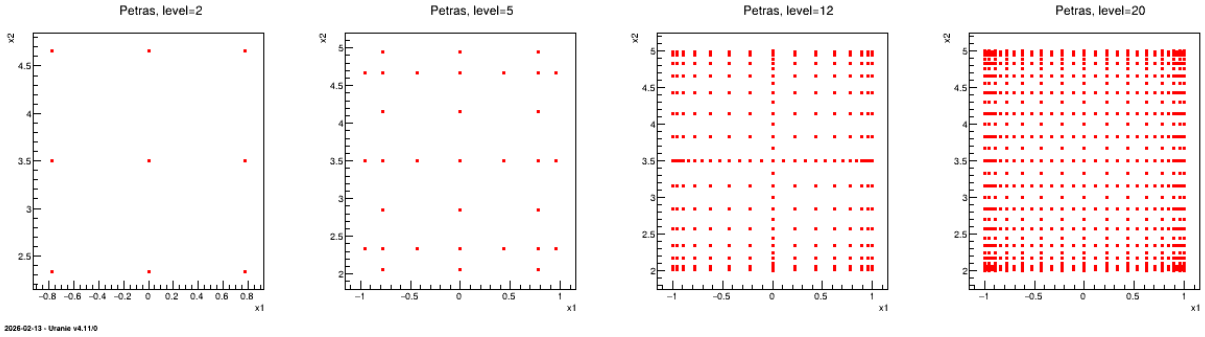


Figure 3.11: Comparison of design-of-experiments made with Petras algorithm, using different level values, when dealing with two uniform attributes.

In the case of regular sequence, the selected sequence is specified with the second argument **option** of the class constructor **TQMC**:

```
TQMC(tds, option, nCalcul )
```

First, a pointer to a `TDataServer` is constructed

```
tdsqmc = DataServer.TDataServer("tdsQMC", "Test for qMC")
tdsqmc.addAttribute(DataServer.TAttribute("x_{1}", -1.0 , 2.))
tdsqmc.addAttribute(DataServer.TNormalDistribution("x_{2}",  0.0 , 3.))
tdsqmc.addAttribute(DataServer.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

```
qmc = Sampler.TQMC(tdsqmc, "sobol", 300)
qmc.generateSample()
```

3.5 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
tds = DataServer.TDataServer("TDSField", "Weight as Field")
tds.addAttribute(DataServer.TAttribute("x", 1, 51))
tds.addAttribute(DataServer.TAttribute("y", 1, 51))
tds.addAttribute(DataServer.TAttribute("Weight"))
```

(continues on next page)

(continued from previous page)

```
# Gaussian Sampler Field
tsf = Sampler.TSamplerField(tds)  # 1
tsf.SetScaleFactor(10.0)
tsf.SetRandomFunction(1000)
tsf.setVariogram("gauss")
tsf.SetVariance(1.0)

tsf.generateSample()  # 2

# Graphic
canvas = ROOT.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")  # 3
```

1. 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(tds, option = "Gauss")`**.
2. The `generateSample()` method generates a sample which will be saved in the `TDataServer`.
3. This step, along with the other drawing method few line earlier, gives the results shown in Figure 3.12.

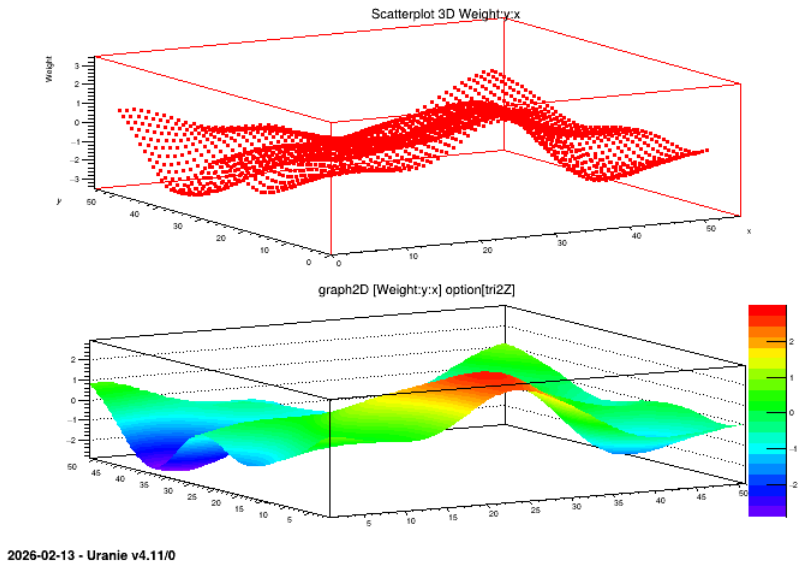


Figure 3.12: Gaussian Random Field

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 3.13 and for Sine Cardinal in Figure 3.14.

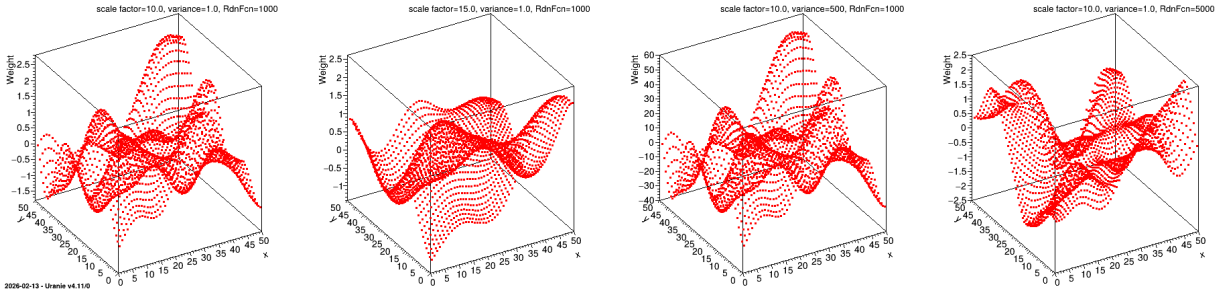


Figure 3.13: Gaussian variograms. Several configurations (in terms of scale factor and variance parameters) are shown as well.

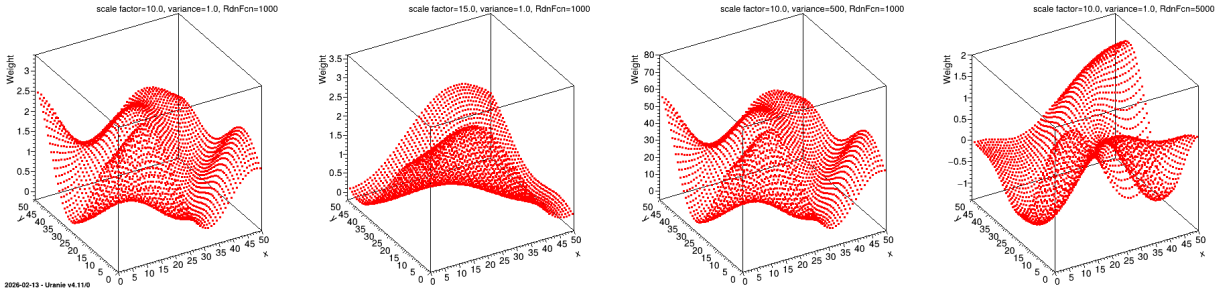


Figure 3.14: Sine cardinal variograms. Several configurations (in terms of scale factor and variance parameters) are shown as well.

3.6 OAT Design

3.6.1 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.

3.6.2 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`**.

3.6.3 `TOATDesign`

3.6.3.1 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:

```
"""
Example of simple OAT DoE
"""
from URANIE import DataServer, Sampler

# step 1
tds = DataServer.TDataServer("tdsoat", "Dataserver simple OAT design")
tds.addAttribute(DataServer.TAttribute("x1"))
tds.addAttribute(DataServer.TAttribute("x2"))

# step 2
tds.getAttribute("x1").setDefaultValue(0.0)
tds.getAttribute("x2").setDefaultValue(10.0)

# step 3
oatSampler = Sampler.TOATDesign(tds)

# step 4
use_percentage = True
oatSampler.setRange("x1", 2.0)
oatSampler.setRange("x2", 40.0, use_percentage)

# step 5
oatSampler.generateSample()

# display
tds.scan("*","","colsize=15 col=:2:2:")
```

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 *Multiple sets of nominal values*
- **__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.

3.6.3.2 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 $\left[nominal - \frac{range}{2}; nominal + \frac{range}{2}\right]$. 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 3.3). The sample's distribution is given by the type of the attribute representing the factor (cf. *Introducing the TStochasticAttribute classes*).
- **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 *Construction of a simple OAT design-of-experiments*, 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.

3.6.3.3 Regular mode

In the example of *Construction of a simple OAT design-of-experiments*, if we change the "step 3" to:

```
oatSampler = Sampler.TOATDesign(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 *Macro "samplingOATRegular.py"* for complete code).

3.6.3.4 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. *Introducing the TStochasticAttribute classes*). We thus need to modify the "step 1" of example of *Construction of a simple OAT design-of-experiments* to:

```
# step 1
tds = DataServer.TDataServer("tdsoat", "Data server for simple OAT design")
tds.addAttribute(DataServer.TUniformDistribution("x1", -5.0, 5.0))
tds.addAttribute(DataServer.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 *Construction of a simple OAT design-of-experiments* does not need to be modified. The "step 3", on the other hand, becomes:

```
# step 3
oatSampler = Sampler.TOATDesign(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.

[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 !

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
c = ROOT.TCanvas("can", "can", 10, 32, 1200, 600)
apad = ROOT.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")
```

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 3.15. 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 *Macro "samplingOATRandom.py"*).

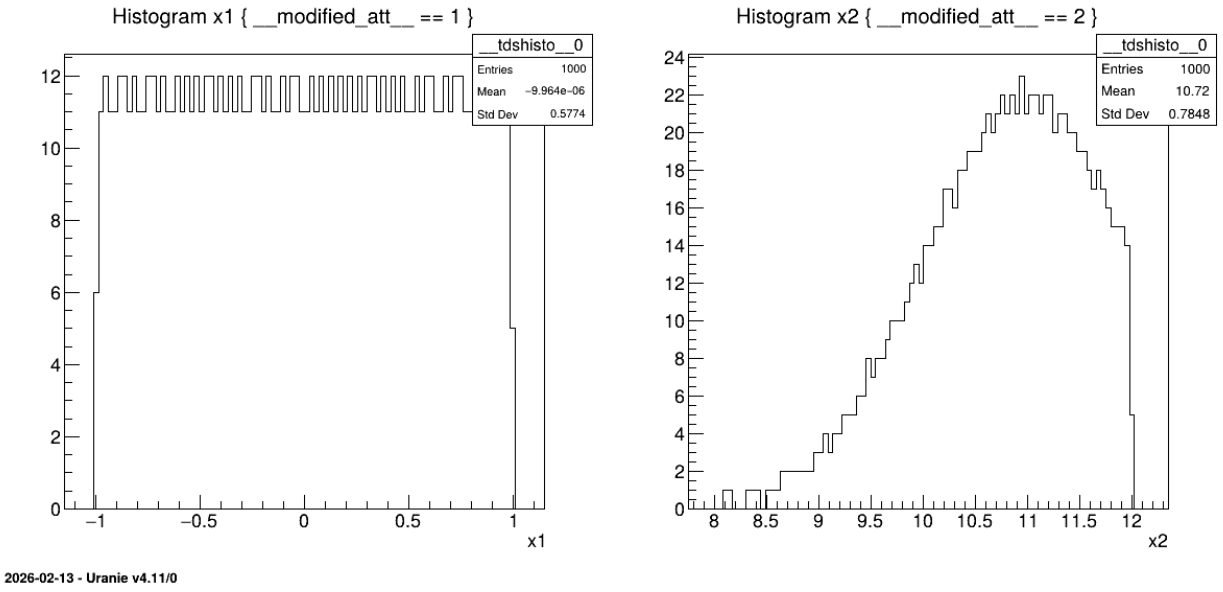


Figure 3.15: Random values for OAT design

3.6.3.5 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. *Data handling* 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 *Construction of a simple OAT design-of-experiments*, the only step requiring modifications is the first one:

```
# step 1
tds = DataServer.TDataServer("tdsoat", "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 *Macro "samplingOATMulti.py"*.

3.6.3.6 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 *Multiple sets of nominal values* 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
use_percentage = True
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 *Macro "samplingOATRange.py"* 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**.

3.6.3.7 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::setRange` for details.

3.7 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 3.16 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.

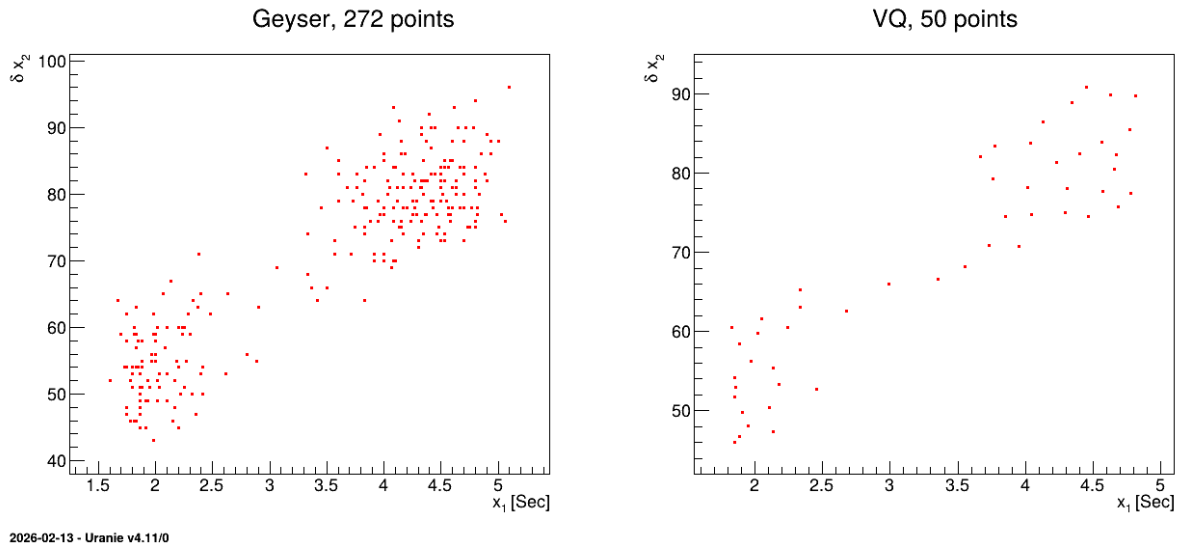


Figure 3.16: Example of a dataset reduction (the geyser one) using the NeuralGas algorithm, to go from 272 points (left) to 50 one (right)

Here is an example of how to use the neuralgas algorithm to reduce a database.

```
c = ROOT.TCanvas("Can", "Can", 10, 32, 1300, 600)  # 1
c.Divide(2, 1)

tdsGeyser = DataServer.TDataServer("tdsgeyser", "Neural Gas for Geyser")
tdsGeyser.fileDataRead("geyser.dat")

tvq = Sampler.TNeuralGas(tdsGeyser, "", 50)  # 2
tvq.setDrawProgressBar(False)
```

(continues on next page)

(continued from previous page)

```
tdsng = tvq.getSubSample("loop=20")  # 3

c.cd(1)
tdsGeyser.draw("x2:x1")
ROOT.gPad.GetPrimitive("__tdshisto__0").SetTitle("Geyser, 272 points")  # 4

c.cd(2)
tdsng.draw("x2:x1")
ROOT.gPad.GetPrimitive("__tdshisto__0").SetTitle("VQ, 50 points")
```

Construction of the plot

1. Creation of `TCanvas` subdivided into two pieces to compare results.
2. Construction of the neuralgas object from the provided database
3. Get the sub-sample as a new dataserver, after looping 20 the algorithm
4. Access the latest histogram drawn on current pad, to change its title

CHAPTER

FOUR

THE LAUNCHER MODULE

4.1 Introduction.md

4.1.1 Overview of a simple case

4.1.1.1 Presentation of the problem

$$y = f(x) = \frac{2\pi T_u \left(H_u - H_l\right)}{\ln(\frac{r}{r_\omega}) \left[1 + \frac{2LT_u}{\ln(\frac{r}{r_\omega}) r_\omega^2 K_\omega} + \frac{T_u}{T_l}\right]} \tag{4.1}$$

4.2 External Code

4.2.1 Code input and output files

4.2.1.1 Input files

4.2.1.1.1 Creation of input files without model files

4.3 Distribution

4.3.1 Multi-step remote launching to clusters

CHAPTER

FIVE

THE MODELER MODULE

This is a description of the Modeler module whose main goal is to produce a surrogate model from a database stored in a `TDataServer`.

5.1 Introduction

This is a test: The Modeler module

The Modeler module discusses the generation of surrogate models 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 models, 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 *The TLinearRegression Class*.

5.2 The `TLinearRegression` Class

When using the `TLinearRegression` class, one assumes that there is only one output variable and at least one input variable.

5.3 Chaos polynomial expansion

5.3.1 Nisp in a nutshell

5.3.1.1 The integration method

5.4 The kriging method

5.4.1 Construction of a kriging model

5.4.1.1 Example: construction of a simple Kriging model

5.4.1.2 Deterministic trend and bayesian prior

5.4.1.3 Choice of the initial point of the optimisation

5.4.2 Usage of a Kriging model

5.4.2.1 Prediction of a new data set, one-by-one approach

5.4.2.2 Prediction of a new data set, global approach

CHAPTER

SIX

THE SENSITIVITY MODULE

6.1 The Sobol method

6.1.1 Computation of Sobol's sensitivity indices

CHAPTER

SEVEN

THE OPTIMIZER MODULE

7.1 Function optimisation

7.1.1 Rosenbrock function

7.1.1.1 Presentation of the problem

$$f(x_1, x_2) = a \times (x_2 - x_1^2)^2 + b \times (1.0 - x_1)^2 \tag{7.1}$$

CHAPTER

EIGHT

THE RELAUNCHER MODULE

8.1 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 *Macros Relauncher*.

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 8.1.

8.2 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 *Rosenbrock function*). 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.

```
#!/usr/bin/env python3

import ROOT

### user evaluation function
def rosenbrock(x, y) :
    d1 = (1-x)
    d2 = (y-x*x)
    return [d1*d1 + d2*d2,]

### study section

# problem variables
itemvar = [
    ROOT.URANIE.DataServer.TAttribute("x", -3.0, 3.0),
    ROOT.URANIE.DataServer.TAttribute("y", -4., 6.),
]
ros = ROOT.URANIE.DataServer.TAttribute("rose")

# user evaluation function
eval = ROOT.URANIE.Relauncher.TPythonEval(rosenbrock)
for a in itemvar :
    eval.addInput(a)
eval.addOutput(ros)

# resources
run = ROOT.URANIE.Relauncher.TSequentialRun(eval)
#run = ROOT.URANIE.MpiRelauncher.TMpiRun(eval)
run.startSlave()
if run.onMaster() :
    # data server
    tds = ROOT.URANIE.DataServer.TDataServer("rosopt", "Rosenbrock Optimisation")
    for a in itemvar :
        tds.addAttribute(a)

    # optimisation
    algo = ROOT.URANIE.Reoptimizer.TVizirGenetic()
    study = ROOT.URANIE.Reoptimizer.TVizir2(tds, run, algo)
    study.addObjective(ros)
    study.solverLoop()

    # save results
    tds.exportData("pyrosenbrock.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.

```
#  both side definitions
...

run.startSlave()  # slave is evaluation side
if run.onMaster() :
    # master is study side
    # study definitions
    ...

    run.stopSlave()
    pass
```

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 *The Reoptimizer module*). The colored arrows in Figure 8.1 show the allowed associations of assessors and runners (to respect, for instance, the thread-safe properties of the assessors).

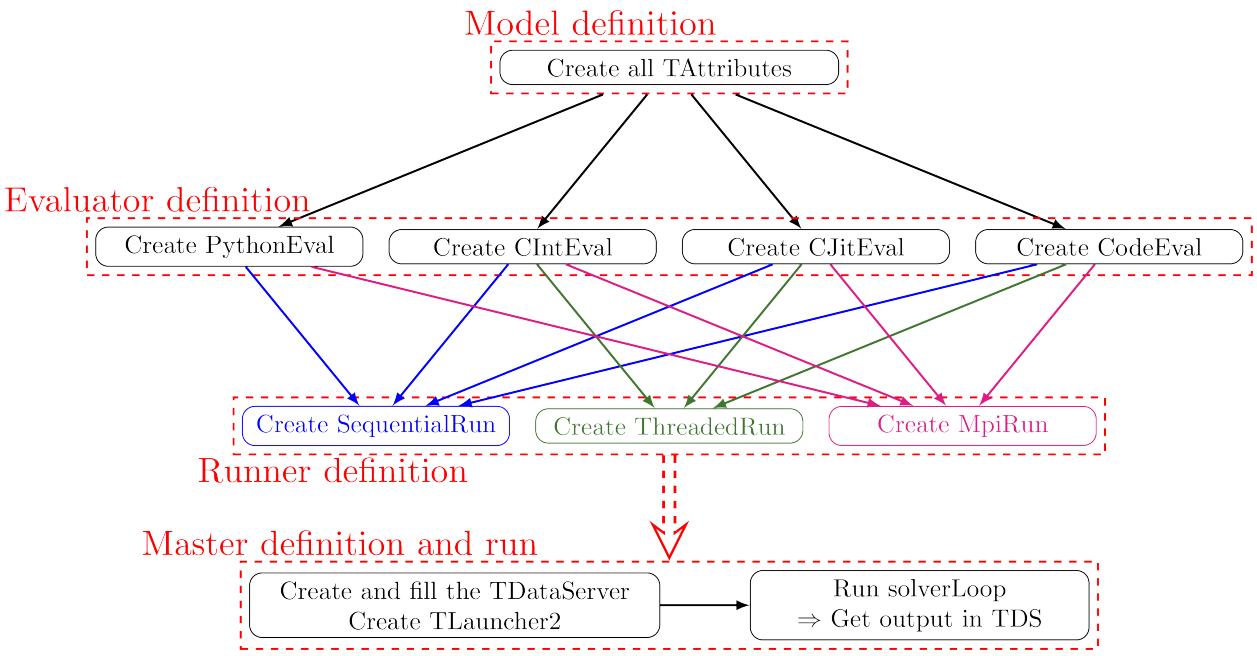


Figure 8.1: Schematic description of the needed steps to define a relauncher procedure

The next three sections will successively pass through the different level of abstraction, starting from the bottom level to the top one.

8.3 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 8.2 that displays the hierarchy of class starting from the inheritance of `TEval`.

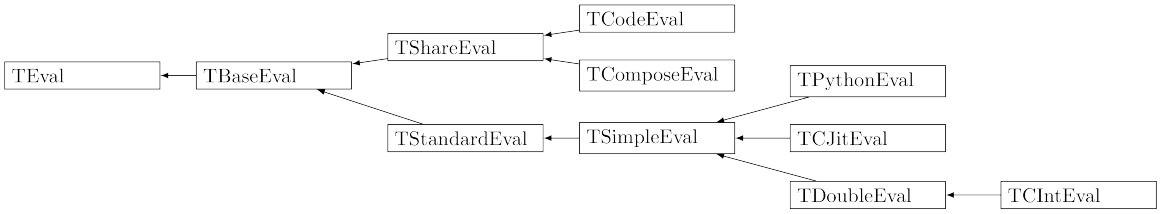


Figure 8.2: Hierarchy of classes and structures for the evaluation part of the Relauncher module.

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 *TMaster* as these specification will be done directly on the master object.

8.3.1 TPythonEVal

8.3.1.1 function convention

The Python function, that can be used with Uranie, receives item values directly in the function parameters and return an output list (even if they are only one value). The rosenbrock example is given below.

```
def rosenbrock(x, y) :
    d1 = (1-x)
    d2 = (y-x*x)
    return [d1*d1 + d2*d2,]
```

`TThreadedRun` should now be supported and `TMpiRun` works but exits badly: the C++ destructor is not called, resulting in an incorrect MPI exit (MPI_Finalize is not call). Put the `stopSlave` method call as last instruction to avoid that evaluation processes stop before master process.

The `TPythonEval` is also able to deal with vectors and strings as inputs and outputs. If the way to deal with string is transparent, vector are just simple plain list of real.

8.3.1.2 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 *Relauncher abstraction levels*:

```
# problem variables
x = DataServer.TAttribute("x", -3.0, 3.0)
y = DataServer.TAttribute("y", -4., 6.)
ros = DataServer.TAttribute("rose")

# user evaluation function
eval = Relauncher.TPythonEval(rosenbrock)
eval.addInput(x)  # Adding attribute in the correct order
eval.addInput(y)
eval.addOutput(ros)
```

8.3.2 `TCIntEval` and `TCJitEval`

These classes are generaly used when you use the cling root interpreter with Uranie. So they are not detailed here. Their documentation can be find in the C++ version of this documentation, but their uses are very similar to a `TPythonEval` class.

Using C function in python can be done with the ROOT cling interpreter and the `TCIntEval`. Suppose you have a C rosenbrock function defined in the `UserFunctions.C` file as follow.

```
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;
}
```

To use this function in Uranie, you need to load the script, and define the `TCIntEVal` (a `TCJitEval` does not work in python interpreter)

```
ROOT.gROOT.LoadMacro("UserFunctions.C")
fun = Relauncher.TCIntEval("rosenbrock")
```

It is also possible to simply use a C++ function in python, through the ROOT cling interpreter out of the Relauncher paradigm (for single-purpose estimation for instance). This discussion is already covered in *Use a C++ function interactively*.

8.3.3 `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.

8.3.3.1 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.

8.3.3.2 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 *Dealing with attributes*).
>
> 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 *Macro "relauncherCodeMultiTypeKeyEmptyVectors.py"* or considered as an error in *Macro "relauncherCodeMultiTypeKeyEmptyVectorsAsFailure.py"* 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 *Dealing with attributes*).

8.3.3.3 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
finp1 = Relauncher.TFlatScript("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
kinp1 = Relauncher.TKeyScript("input_rosenbrock_with_keys.dat")
kinp1.addInput(x, "x")  # Adding attributes in the correct order, one-by-one
kinp1.addInput(y, "y")
```

Once done, the input files are provided to the `TCodeEval` object, using the `addInputFile` method, as shown below:

```
# Add to the TCodeEval
code = Relauncher.TCodeEval("rosenbrock -r")  # put "rosenbrock -k" instead for key
code.addInputFile(finp1)  # put kinp1 instead for key
```

8.3.3.4 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 *Creation of input files without model files* and in *Objective*. 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
fout = Relauncher.TFlatResult("_output_rosenbrock_with_values_rows.dat")
fout.addOutput(ros)

# Or Input File Key format case
kout = Relauncher.TKeyResult("_output_rosenbrock_with_keys.dat")
kout.addOutput(ros, "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
```

8.3.4 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.

8.3.4.1 `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 *Macro "relauncherComposeMultitypeAndReadMultiType.py"*.

8.4 `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 8.3. 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.

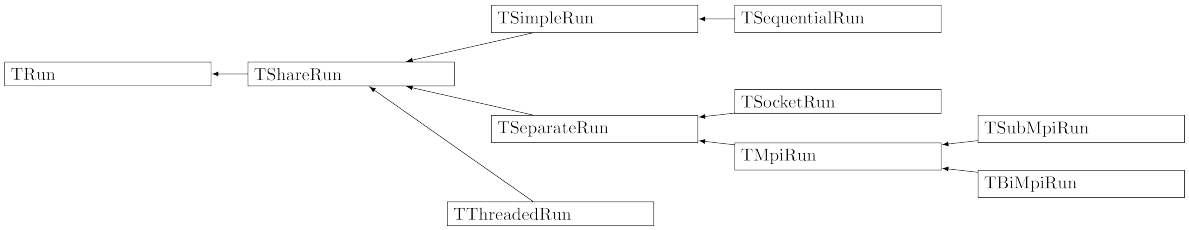


Figure 8.3: Hierarchy of classes and structures for the runner part of the `Relauncher` module.

8.4.1 `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
srun = Relauncher.TSequentialRun(code)
```

8.4.2 `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
trun = Relauncher.TThreadedRun(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 *Relauncher abstraction levels*) 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:

```
import ROOT
ROOT.EnableThreadSafety()
```

8.4.3 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.
- `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. In the specific case of python, where ROOT and python (though the garbage collector approach) are arguing to destroy objects, a specific line (`ROOT.SetOwnership(run, True)`) has to be added, as discussed in *Macro*.

`TMpiRun` constructor has one argument, a pointer to a `TEval` object.

```
# Creating the threaded runner
mrun = Relauncher.TMpiRun(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 python RosenbrockMacro.py
```

Here, we launch root on 8 cores (`-n 8`). 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.

> **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.

8.4.3.1 `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 (`TPythonEval`) 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).

8.5 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.

8.5.1 Dealing with attributes

Disregarding the kind of master considered (the `TLauncher2`, discussed briefly below or those introduced in the optimisation part, in *Problem definition* 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 *Macros Relauncher*. 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 *Macros Relauncher*.
- 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.

8.5.2 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`.

8.5.2.1 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 *Various file format* and illustrated in a use-case macro *Macro "relauncherCodeMultiTypeKeyEmptyVectorsAsFailure.py"* 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 *Dealing with attributes* where all the methods are inherited from the `TMaster` class, the method `setSaveError(TDataServer *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 *Various file format* and illustrated in a use-case macro *Macro "relauncherCodeFlowrateSequentialFailure.py"* and when an empty vector is interpreted as a missing information because of the wrong output file formatting, see *Macro "relauncherCodeMultiTypeKeyEmptyVectorsAsFailure.py"*.

CHAPTER

NINE

THE REOPTIMIZER MODULE

9.1 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 [Bla17]), 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 *Macros Reoptimizer*.

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.

9.1.1 local optimizer

A local solver proposes mono-objective optimisation with or without constraints. It uses the NLopt library [Joh] 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.

9.1.2 global optimizer

A global solver proposes mono and multiobjective optimisation, potentially with constraints. It uses the Vizir library [Arn10] and offers different solvers.

9.1.3 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 [Bla17].

9.2 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 *The Relauncher module* 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 *The Relauncher module*). 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 *Relauncher abstraction levels* 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
    tds = DataServer.TDataServer("nloptDemo", "Param de l'opt nlopt pour la barre")
    tds.addAttribute(x)
    tds.addAttribute(y)

    # Defining the optimisation condition
    solv = Reoptimizer.TNloptCobyla()  # algorithm

    # Create the single-objective constrained optimizer master
    opt = Reoptimizer.TNlopt(tds, runner, solv)
    # ... + objective, constraint...

    # Create the dataserver in which all computation will be stored
    trc = DataServer.("allevents", "dataserver containing all events")
    opt.setTrace(trc)  # pass the dataserver to the master

    opt.solverLoop()  # perform the optimisation
```

9.2.1 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.

9.2.2 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 9.1.

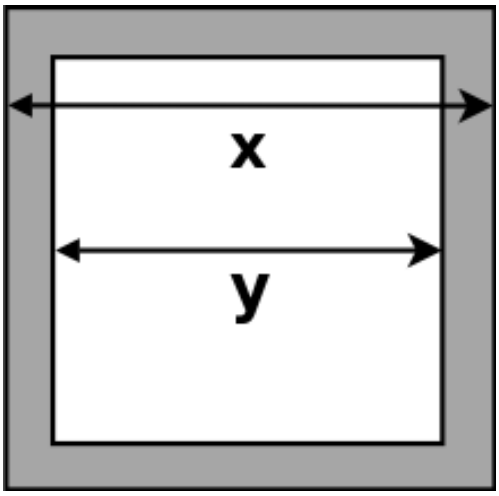


Figure 9.1: Hollow Bar

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:

```
#!/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
```

(continues on next page)

(continued from previous page)

```
    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`

9.3 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 *Macro "reoptimizeHollowBarCode.py"* with a specification with respect to the way it was described in *Sizing of a hollow bar example problem*: 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).

9.3.1 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 Python, it could look like this:

```
import numpy
...
opt = Reoptimizer.TNlopt(tds, runner, solv)
...
p = numpy.array([0.2, 0.3])
opt.setStartingPoint(len(p), p)
```

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.

9.3.2 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.

9.4 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 [Bla17]) 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 *Macro "reoptimizeHollowBarCodevizir.py"*.

9.4.1 A step-by-step description of Vizir

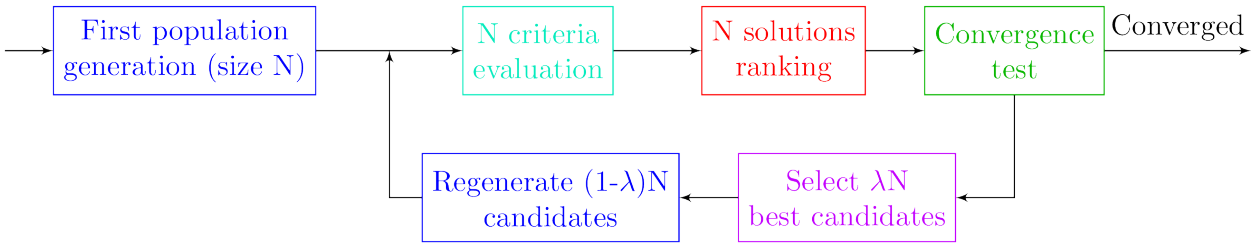


Figure 9.2: Schematic description of the requested steps of an optimisation procedure once this one is performed with Vizir

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 9.2) is followed by the ranking of all the candidates (red step in Figure 9.2).
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 [Bla17].

9.4.2 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).

9.4.3 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.

9.4.3.1 TVizirGenetic

The `TVizirGenetic` (whose principle and vocabulary is define in [Bla17]) 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%).

9.4.3.2 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.

9.4.3.3 TVizirSimplex

The `TVizirSimplex` have not any specific configuration method.

9.4.3.4 Many objectives methods

The global solvers discussed here have dedicated methods to call many-objective algorithms. An example of implementation can be shown in *Macro "reoptimizeHollowBarVizirMoead.py"*.

- `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

TEN

THE METAMODEL OPTIMIZATION MODULE

10.1 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.

10.2 Efficient Global Optimization

10.2.1 Introduction

EGO [JSW98] 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.

10.2.1.1 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.

10.2.2 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 *The Relauncher module* and at *The Reoptimizer module* for a better understanding. As it uses a kriging model, it is also suggested to look at *The kriging method*.

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.

10.2.2.1 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`

10.2.2.2 TEgoModeler

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

10.2.2.2.1 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 *The kriging method* for details on possible parameter values.

10.2.2.3 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 `TEgoHjStdSolver`

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

ELEVEN

THE CALIBRATION MODULE

Abstract This chapter presents the features of the Calibration module of Uranie - version v4.11.0. The *namespace* of this library is **URANIE::Calibration**.

11.1 Introduction

This section presents different calibration methods that are provided to help achieve an accurate estimation of the parameters of a model with respect to data (either from experiment or from simulation). The methods implemented in Uranie are ranging from point estimation to more advanced Bayesian techniques and they mainly differ in the hypotheses they rely on.

They are all gathered in the **libCalibration** module. The namespace of this library is **URANIE::Calibration**. Each technique discussed later on is theoretically introduced in [Bla17] along with a general discussion on calibration and particularly on its statistical interpretation.

The reference data will be compared with model predictions, where the model is a mathematical function $\mathbf{f}_\theta : \mathbb{R}^{n_X} \rightarrow \mathbb{R}^{n_Y}$. From now on and unless otherwise specified 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 observations will then be written y and the predictions of the model $f_\theta(\mathbf{x})$).

In addition to the problem-dependent input vector, the model also depends 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 values θ is called a **configuration**.

The rest of this section introduces the available distances and likelihoods used to compare observations with model predictions, in *Distances and likelihoods used to compare observations and model predictions* while the methods are discussed in their own sections. The already predefined calibration methods proposed in the Uranie platform are listed below:

- *Minimisation techniques*
- *Analytical linear Bayesian estimation*
- *Approximate Bayesian Computation techniques (ABC)*
- *Markov chain Monte Carlo approach*

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 11.1 and is discussed a bit here to explain the two main classes from which all other classes are derived and the corresponding shared functions used throughout the methods. One can see this organisation with the two sets of classes: those inheriting from the `TCalibration` class and those inheriting from `TDistanceLikelihoodFunction` class. The former are the different methods that have been developed to calibrate a model with respect to the observations and each method will be discussed in the upcoming sections. Whatever the method under consideration, it always includes a distance or a likelihood 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 *Distances and likelihoods used to compare observations and model predictions*.

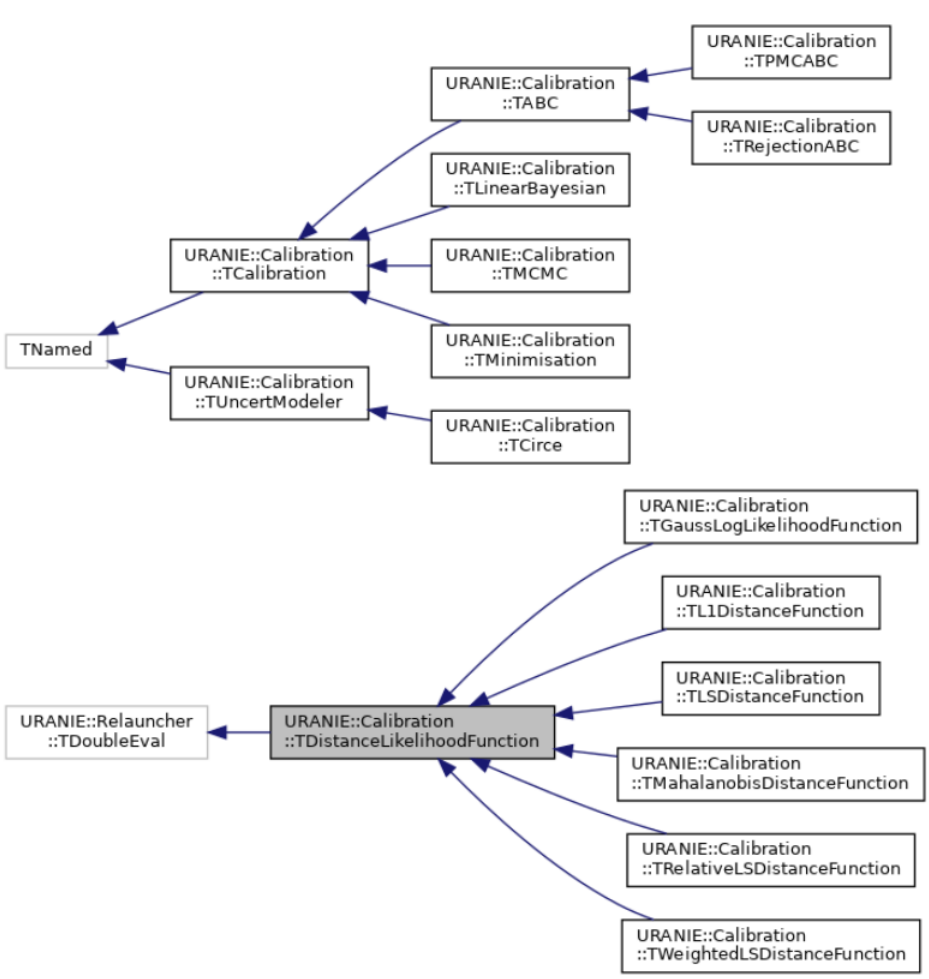


Figure 11.1: Hierarchy of classes and structures out of Doxygen for the **Calibration** module

Although CIRCE is not strictly a calibration method, it has been included in this section because it relies on approaches presented here. Indeed, the idea behind this method is to quantify the uncertainty of a given quantity by multiplying it by a Gaussian random variable whose standard deviation must be calibrated. A final section therefore introduces this method.

- *CIRCE method*

The next section focuses on the distances and likelihoods already implemented in Uranie, which can be used directly within the calibration methods.

11.1.1 Distances and likelihoods used to compare observations and model predictions

There are many ways to quantify the agreement between the reference observations and the model predictions, given a parameter vector θ. Depending on the framework adopted (deterministic or Bayesian), different tools are required. In a deterministic setting, a distance is used to measure how far the prediction is from the observation. In contrast, within the Bayesian framework, the analogous concept is the likelihood, a function that evaluates the probability of observing the data given a parameter vector.

Since the number of variables n_Y used to perform the calibration can be greater than one, it might be useful to introduce the coefficients $\{\omega_j\}_{j\in[1,n_Y]}$ to weight the contribution of each variable relative to the others. The following lists the distance functions available in Uranie:

- 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 squares distance function: $d(\mathbf{y}, \mathbf{f}_\theta(\mathbf{x})) = \sum_{j=1}^{n_Y} \sqrt{\omega_j \sum_{i=1}^{n} (\mathbf{y}_i^j - \mathbf{f}_\theta(\mathbf{x})_i^j)^2}$;
- Relative least squares distance function: $d(\mathbf{y}, \mathbf{f}_\theta(\mathbf{x})) = \sum_{j=1}^{n_Y} \sqrt{\omega_j \sum_{i=1}^{n} \left(\frac{\mathbf{y}_i^j - \mathbf{f}_\theta(\mathbf{x})_i^j}{\mathbf{y}_i^j} \right)^2}$;
- Weighted least squares distance function: $d(\mathbf{y}, \mathbf{f}_\theta(\mathbf{x})) = \sum_{j=1}^{n_Y} \sqrt{\omega_j \sum_{i=1}^{n} \psi_i^j \times (\mathbf{y}_i^j - \mathbf{f}_\theta(\mathbf{x})_i^j)^2}$, where the coefficients $\{\psi_i^j\}_{i \in [1,n]}$ are associated with the j-th variable and are used to weight each observation with respect to the others;
- Mahalanobis distance function: $d(\mathbf{y}, \mathbf{f}_\theta(\mathbf{x})) = \sum_{j=1}^{n_Y} \sqrt{\omega_j (\mathbf{y}^j - \mathbf{f}_\theta(\mathbf{x})^j)^T \Sigma^{-1} (\mathbf{y}^j - \mathbf{f}_\theta(\mathbf{x})^j)}$ where Σ is the covariance matrix of the observations.

Regarding the likelihood functions already implemented, only the Gaussian log-likelihood for independent parameters is available, as it is the most commonly used. Its expression follows:

- Gaussian log-likelihood for independent parameters: $\textit{log-}\mathcal{L}(\theta|\mathbf{x}, \mathbf{y}) = -\frac{1}{2} \sum_{j=1}^{n_Y} \sum_{i=1}^{n} \left(\log\left(2\pi \left(\sigma_i^j\right)^2\right) + \left(\frac{\mathbf{y}_i^j - \mathbf{f}_\theta(\mathbf{x})_i^j}{\sigma_i^j} \right)^2 \right)$ where the coefficients $\{\sigma_i^j\}_{i \in [1,n]}$ are the standard deviations of each observation associated to the j-th variable.

If needed, it is still possible to define a custom likelihood (or distance).

For more details on the implementation of the distances and likelihoods, or on how to implement your own, see *Defining data, distance and likelihood functions*.

11.2 Calibration classes, distance and likelihood functions, observations and model

This section introduces the common elements of all analyses in the **Calibration** module. Indeed the methods discussed hereafter will be using the same architecture and will require a common set of items, listed below:

- a model has to be set up as this is what one wants to calibrate. It can come either as a `Relauncher::TRun` instance, as a `Launcher::TCode` or a **Launcher** function. This part is introduced in *General introduction to data and model definition* (for the general concept and the difference with the usual organisation of model definition) and discussed later on (mainly in relation to the constructor of `TCalibration`-derived objects) in *Common methods of the calibration classes* and in the dedicated section in each method;
- a reference observations have to be defined once and for all so that the model can be run for each newly defined set of parameter values (every new configuration). This part is discussed first in *General introduction to data and model definition* and the way to provide them is also partly discussed in *Defining data, distance and likelihood functions*;
- a distance or likelihood function has to be created, usually within the calibration instance, to quantify how well the model under study reproduces the reference observations. This part is discussed in *Defining data, distance and likelihood functions*;
- a main object has to be created, a calibration method instance, that inherits from the `TCalibration` class. This is discussed in *Common methods of the calibration classes*.

11.2.1 General introduction to data and model definition

All calibration problems will have at least two `TDataServer` objects:

- The reference one, usually called `tdsRef`, contains the observations (both input and output attributes) on which the calibration is to be performed. It is generally read from a simple input file as done below:

```
tdsRef = DataServer.TDataServer("reference", "my reference")
tdsRef.fileDataRead("myInputData.dat")
```

- the parameter one, usually called `tdsPar`, 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 *Minimisation techniques*) 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 *Analytical linear Bayesian estimation*) along with all those that require generating one or more design-of-experiments (see *Approximate Bayesian Computation techniques (ABC)* but also *Markov chain Monte Carlo approach*).

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 *Macros Launcher* and in *Macros Relauncher* as the inputs of the model are coming from two different `TDataServers`, they can be split into two categories:

- the reference ones will only have values from the reference input file `myInputData.dat`, which contains n observations. For each configuration, the model is evaluated n times: the configuration remains fixed, while the reference values are varied across those n observations;
- the parameter ones, on the other hand, change only when moving to a new configuration and remain constant across all n reference evaluations.

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 terms of order, 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
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 particular attention must be paid 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 *Common methods of the calibration classes*.

Finally, all models considered for calibration should have exactly as many outputs (whatever their names are) as the number of outputs 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 or likelihood 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 find three different kinds of information (more can be added if needed, see *Running the estimate*):

- the resulting calibrated value of the parameters (or values depending on the chosen method, as some of them are providing several samples of calibrated configurations) is 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 calibrated configuration;
- the residuals: the difference between the model predictions and the reference data for the n observations, using the *a priori* and *a posteriori* configurations. These are stored in a dedicated `TDataServer` object, called the **EvaluationTDS** (referred to as `tdsEval`) and it is mainly called through the `drawResiduals` method (discussed in *Drawing the residuals*). If one wants to access it, it is possible to get a hand on it by calling the `getEvaluationTDS()` method. The residuals are important to check that the behaviour of the newly performed calibration does not show unexpected and unexplained tendencies with respect to any variables in the defined uncertainty setup, see the dedicated discussion on this in [Bla17].

11.2.2 Defining data, distance and likelihood functions

The different distance and likelihood functions already embedded in Uranie can be found in *Distances and likelihoods used to compare observations and model predictions* and are further discussed from a theoretical point of view in [Bla17]. From the user point of view, on the other hand, every distance or likelihood function inherits from the class `TDistanceLikelihoodFunction`, as can be seen in Figure 11.1, which is purely virtual (meaning that no object can be instantiated as `TDistanceLikelihoodFunction`) and serves several main purposes:

- it loads the reference data once and stores them internally as a vector of vectors (or as a vector of `TMatrixD`, depending on the chosen formalism used to compute the distance or likelihood);
- once done, the following loop is executed as long as new configurations need to be tested (a configuration being defined as a new set of values for vector θ):
  1. it runs the chosen model regardless of the nature of the object (`TRun`, `TCode`, etc.) on the full reference dataset to get the new predictions;
  2. it loads the new model predictions for the configuration under study into a vector of vectors (or as a vector of `TMatrixD` depending on the chosen formalism);
  3. it computes the distance or likelihood using both vectors as statd by the equations in *Distances and likelihoods used to compare observations and model predictions*. 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 or likelihood function, see dedicated discussion in *Creating its own distance or likelihood function*).

From a technical point of view, the `TDistanceLikelihoodFunction` 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 the straightforward use of all **TNlopt** algorithms as well as **Vizir** solutions (see *Minimisation techniques*).

Regardless of the calibration method considered, a distance or likelihood function must be used to compare data and model predictions. This is even true for the `TLinearBayesian` class, which directly computes the analytical posterior distribution, as the residuals are computed both *a priori* and *a posteriori* to assess the improvement of the prediction and their consistency within the uncertainty model (see discussion in [Bla17]). In most cases the object will be constructed with a recommended way (discussed in *Recommended distance and likelihood functions, construction method*). Another possibility is also discussed in *Creating its own distance or likelihood function*

Whatever the situation, once a calibration instance is created (for the sake of generality, we will use an instance named `cal`, of the mock class `TCalClass`, as if it inherited from the `TCalibration` class), the first method to be called is the `setDistance` or `setLikelihood` depending on the framework used (see the corresponding section), as this is the method which defines both the type of distance or likelihood function and the observation dataset.

11.2.2.1 Recommended distance and likelihood functions, construction method

There are several ways to define a distance or a likelihood function. The recommended approach is to use the `setDistance` and `setLikelihood` methods, which are directly available in each calibration class such as `TLinearBayesian` or `TMinimisation`. Alternatively, one can call the `setDistanceAndLikelihood` method from the `TCalibration` class (which is actually invoked by the two previous methods), and is inherited by all calibration classes. However, we recommend using the first two approaches, in order to prevent mistakenly assigning distances to likelihood-based methods, or likelihood functions to distance-based methods. The prototypes discussed here are as follows:

```
setDistanceAndLikelihood(funcName, tdsRef, input, reference, weight="")
setDistance(funcName, tdsRef, input, reference, weight="")
setLikelihood(funcName, tdsRef, input, reference, weight="")
```

It takes up to five arguments, one of which is optional:

1. **funcName**: the name of the distance or likelihood function, corresponding to one of the already implemented one, as discussed in *Distances and likelihoods used to compare observations and model predictions*. The possible choices are
   - "L1" for `TL1DistanceFunction`;
   - "LS" for `TLSDistanceFunction`;
   - "RelativeLS" for `TRelativeLSDistanceFunction`;
   - "WeightedLS" for `TWeightedLSDistanceFunction`;
   - "Mahalanobis" for `TMahalanobisDistanceFunction`;
   - "log-gauss" for `TGaussLogLikelihoodFunction`.
2. **tdsRef**: the `TDataServer` in which the observations are stored;
3. **input**: the input variables stored in the `TDataServer` tdsRef, which must be defined as inputs in the code 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, against which the output of the code or function will be compared. This argument has the usual attribute list format "out1:out2:out3";
5. **weight** (optional): this argument is optional and can be used to define the name of the (single) variable stored in the `TDataServer` tdsRef which, in the case of a `TWeightedLSDistanceFunction`, is used to fill the $\{\psi_i^j\}_{i\in[1,n]}$, *i.e.* the coefficients that weight each observation relative to the others and, in the case of a `TGaussLogLikelihoodFunction`, is used to fill the $\{\sigma_i^j\}_{i\in[1,n]}$, *i.e.* the standard deviations of each observation associated to the j-th variable (see *Distances and likelihoods used to compare observations and model predictions*).

> **Warning**
>
> The number of variables in the **weight** list should match the number of outputs of your code used to calibrate the parameters. If one output is not weighted, a "one" attribute should be added to handle that output, when the other requires an uncertainty model.

Once this method is called, the distance or likelihood function is created and is stored within the calibration object. It may be necessary to access it for certain options, but this is further discussed in *Available options for every distance and likelihood function*.

The following line summarises this construction in a case where an instance `cal` of the fake class `TCalClass` (as if this class was inheriting from the `TCalibration` class) is created.

```
# Define the dataservers
tdsRef = DataServer.TDataServer("reference", "myReferenceData")
# Load the data, both inputs (ref_var1 and ref_var2) and a single output (ref_out1).
tdsRef.fileDataRead("myInputData.dat")
...
tdsPar = DataServer.TDataServer("parameters", "myParameters")
tdsPar.addAttribute(DataServer.TNormalDistribution("par1", 0, 1))
# the parameter to calibrate
...
# Define the model
...
# Create the instance of TCalClass
cal = Calibration.TCalClass()  # Constructor is discussed later on
# Define the least squares distance
cal.setDistance("LS", tdsRef, "ref_var1:ref_var2", "ref_out1")
```

In this fake example, the distance function is the least squares one, and it will use the n values of both inputs **"ref_var1"** and **"ref_var2"** and output **"ref_out1"** stored in `tdsRef` to calibrate the parameter **"par1"**. No observation weight is needed in this case, as least squares method does not require it and as there is only a single output, no variable weight needs to be defined either.

11.2.2.2 Creating its own distance or likelihood function

It is possible to create a custom distance or likelihood function if one wants to use a function not already implemented in Uranie.

11.2.2.3 Available options for every distance and likelihood function

All distance and likelihood function classes inherit from `TDistanceLikelihoodFunction`, and the only difference among them lies in the implementation of the `localeval` function. As a result, a large portion of the code is shared across all the distance and likelihood function objects, including the parts handling their configuration and options.

This subsection gathers all the shared options, which can be configured either through the optional "Option" parameter in several methods and constructors, or by accessing the distance or likelihood function object directly once it has been created. To do so, the user can call the `getDistanceLikelihoodFunction` method, which returns the distance or likelihood function instance stored within the calibration object. The following lines provide an example using an instance `cal` of the mock class `TCalClass` (as if it inherited from the `TCalibration` class).

```
# import rootlogon to get Calibration namespace
from rootlogon import DataServer, Calibration
# Creating the calibration instance from a TDataServer and a Relauncher::TRun
cal = Calibration.TCalClass(tdsPar, runner, 1, "")
# Creating the distance function (a least square one)
cal.setDistance("LS", tdsRef, "logRe:logPr", "logNu")
# Retrieving the instance to be able to change options
dFunc = cal.getDistanceLikelihoodFunction()
```

11.2.2.3.1 Defining the variable weights

When multiple variables are used to compare predictions with observations, it may be desirable to weight their respective contributions to the distance. This is precisely the purpose of the ω_j variable described in *Distances and likelihoods used to compare observations and model predictions*. This option is not relevant for the Gaussian log-likelihood, since the "contribution" of the variables are already weighted by the standard deviation (through the uncertainty on the data). For

all distances, two different methods are available to define some variable weights, both called `setVarWeights`. The only difference lies in their prototype :

```
# Prototype 1 with array defined as
# wei = np.array([0.5, 0.4])
setVarWeights(len(wei), wei)
# Prototype 2 with vectors defined as
# wei = ROOT.std.vector['double']([0.5, 0.4])
setVarWeights(wei)
```

The idea behind these two prototypes is to have a way to control the number of elements in the array of doubles, 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 matches the number of output variables the weights are initialised; otherwise, the program will crash.

11.2.2.3.2 Dumping all the estimates

Whatever the construction and evaluation type (either based on **Launcher** or **Relauncher**), you might want to keep track of all evaluations across all the reference observations and not only the global distance or likelihood. This is possible by calling the `dumpAllDataservers` method of `TDistanceLikelihoodFunction`:

```
dumpAllDataservers()
```

This method takes no arguments and sets a boolean to true (by default it is set to false) which implies that every configuration will be dumped as an ASCII file that will be named according to the following convention: `Calibration_testnumber_XX.dat` where `XX` is the configuration number for the under-study analysis.

> **Warning**
>
> This option might dump a very large number of files which can fill your local disk (for instance, if you are testing functions where the limit on the number of configurations is not strictly enforced).
>
> When used with a runner architecture (**Relauncher**-type evaluator), the parameters are not kept by default in the dataserver (they are defined as `ConstantAttribute`). To get the required behaviour (i.e., including the parameter values in the ASCII file), the user should also call the `keepParametersValue` method of `TDistanceLikelihoodFunction`:
>
> ```
> keepParametersValue()
> ```
>
> This function, which takes no argument, simply sets to true the final argument of all `TLauncher2` calls to `addConstantAttribute`.

11.2.2.3.3 Defining the observations covariance matrix

In the case where the observations are correlated with a known covariance structure, this covariance can be specified as a `TMatrixD` using the following method:

```
setObservationCovarianceMatrix(mat)
```

This method, and the ability to define a covariance structure between observations, is only relevant when the distance function is an instance of the `TMahalanobisDistanceFunction`.

11.2.2.3.4 Specifying 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 appears here, because the distance or likelihood function is the place where the model is used to estimate the new predictions for a newly set parameter vector (see *Defining data, distance and likelihood functions*). This option allows not to use the usual *TLauncher* object in order to use one of the other instances. This is done in a single function:

```
changeLauncher(tlcName)
```

The only argument of this method is a `TString` object which is used to select one of the two other instances 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 a usual `TLauncher` object to another one chosen by the user. Among the possible solutions one can set

- `TLauncherByStep`: if the user wants to break down the launching process, decomposing it 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 setup, such as the CEA clusters. This is based on the `libssh` library, see *Multi-step remote launching to clusters* for more details.

Finally, the default option for the `run` method of any `TLauncher`-inheriting instance is **"noIntermediateSteps"**, which prevents the safe writing of the results into a file every five estimates. Two methods can be applied on `TDistanceLikelihoodFunction` objects to change these options:

```
# Adding more options to the code launcher run method
addCodeLauncherOpt(opt)
# Change the code launcher run method options
changeCodeLauncherOpt(opt)
```

The first one keeps the default options and adds more 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 allows restarting from scratch in order to define the options as chosen by the user.

11.2.2.3.5 Dumping the distance details

In the case where one does not understand or trust the way the distance or likelihood is computed, it is possible to dump the details of their computation (warning: this is very verbose). This can be done by calling the `dumpDetails`, whose prototype is straightforward:

```
dumpDetails()
```

11.2.3 Common methods of the calibration classes

All calibration classes that derive from `TCalibration` share many common methods, and their organisation has been streamlined as much as possible. This section describes the shared code to avoid repetition in the later sections, which cover each specific calibration method (from *Minimisation techniques* to *Markov chain Monte Carlo approach*).

The subsections from *Construction with a TRun* to *Construction for a function with the Launcher architecture* discuss the different constructors, explaining the various ways the model can be provided and their implications while *Running the estimate* provides a glimpse of the shared possible options that can be defined and their purpose. Finally, *Drawing the parameters* and *Drawing the residuals* introduce the drawing methods in principle (though illustrations are postponed to dedicated sections of this documentation).

> **Warning**
>
> As with some earlier discussions, the following methods are common to all classes inheriting from `TCalibration`. For illustration, the examples will use an instance `cal` of the fictitious class `TCalClass` (as if it inherited from `TCalibration` class). A more realistic syntax can be found in the dedicated sections (from *Minimisation techniques* to *Markov chain Monte Carlo approach*).

Calibration classes can generally be constructed in four different ways, depending on how the model has been specified. To estimate how close the new set of parameter values (the **configuration**) produces output that are close to the reference, one needs to be able to run the model on the input variables of the reference dataset. The input variables of the reference dataset are not the only input variables 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 illustrates, using the **fictitious** class `TCalClass`, how calibration objects can be constructed.

11.2.3.1 Construction with a `TRun`

This constructor uses the **Relauncher** architecture. This approach provides a simple way to change the evaluator (e.g., switching from a C++ function to a Python function or another code). It also allows the use of either a sequential approach (for single-thread execution) or a threaded approach (to distribute estimates locally). This approach is partly discussed in *The Relauncher module*.

In this case, the constructor has the following form:

```
# Constructor with a runner
TCalClass(tds, runner, ns=1, option="")
```

It takes up to four arguments, two of which are optional:

1. **tds**: a `TDataServer` object containing one attribute for each parameter to be calibrated. This is the `TDataServer` object called `tdsPar`, defined in *General introduction to data and model definition*.
2. **runner**: a `TRun`-inheriting instance that contains all the model information and whose type determines how the estimates are distributed: it can either be a `TSequentialRun` instance or `TThreadedRun` for distributed computations.
3. **ns** (optional): the number of samples to be produced. This parameter is only relevant for methods that return multiple configurations (either multiple solutions to the minimisation problem or samples from the posterior distribution). It is not used in local minimisation with single-point initialisation, nor in linear Bayesian analysis (see *Analytical linear Bayesian estimation*). The **default** value is 1.
4. **option** (optional): the options that can be applied to the method. Options common to all calibration classes (those defined in the `TCalibration` class) are discussed in *Running the estimate*.

A crucial step in this constructor is the creation of the instance that inherits from `TRun`. As mentioned earlier, its type determines how the estimates are distributed. To construct such an object, one needs to provide an evaluator (the available evaluators are described in detail in *TEval*).

Using the formalism introduced in *Recommended distance and likelihood functions, construction method*, a model instance based on the code `Foo`, initialised with `TCIntEval`, can be created as shown below. This example assumes the model takes three inputs (**"ref_var1"**, **"par1"**, **"ref_var2"**) and it produces a single output, which is compared with the reference output **"ref_out1"** via the distance or likelihood function (see *Recommended distance and likelihood functions, construction method*).

```
# Define the dataservers
tdsRef = DataServer.TDataServer("reference", "myReferenceData")
# Load the data, both inputs (ref_var1 and ref_var2) and a single output (ref_out1).
```

(continues on next page)

(continued from previous page)

```
tdsRef.fileDataRead("myInputData.dat")
...
TDataServer *tdsPar = new TDataServer("parameters", "myParameters")
tdsPar.addAttribute(DataServer.TNormalDistribution("par1", 0, 1))  # parameter to␣
↪calibrate
...
# Define the model if a function Foo is loaded through ROOT.gROOT.LoadMacro(...)
# for which x[0]=ref_var1, x[1]=par1 and x[2]=ref_var2 and y[0] is the model␣
↪prediction
model = Relauncher.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
out = DataServer.TAttribute("out")
model.addOutput(out)
# Define a sequential runner to be used
runner = Relauncher.TSequentialRun(model)
...
# Create the instance of TCalClass
ns = 1
cal = Calibration.TCalClass(tdsPar, runner, ns, "")
```

11.2.3.2 Construction with a `TCode`

This constructor uses the **Launcher** architecture. Unlike the **Relauncher** approach, it is specifically designed for code-based models.

In this case, the constructor has the following form:

```
# Constructor with a TCode
TCalClass(tds, code, ns=1, option="")
```

It takes up to four arguments, two of which are optional:

1. **tds**: a `TDataServer` object containing one attribute for each parameter to be calibrated. This is the `TDataServer` object called `tdsPar`, defined in *General introduction to data and model definition*.
2. **code**: a `TCode` instance containing the output file (or files) listing the output attributes while all input attributes are associated with input files using the standard methods (`setFileKey`, `setFileFlag`, etc.).
3. **ns** (optional): the number of samples to be produced. This parameter is only relevant for methods that return multiple configurations (either multiple solutions to the minimisation problem or samples from the posterior distribution). It is not used in local minimisation with single-point initialisation, nor in linear Bayesian analysis (see *Analytical linear Bayesian estimation*). The **default** value is 1.
4. **option** (optional): the option that can be applied to the method. Options common to all calibration classes (those defined in the `TCalibration` class) are discussed in *Running the estimate*.

Unlike the runner-based constructor, this construction does not define how the computation is performed. By default, the `run` method is called for each configuration with the following options:

- **noIntermediateSteps**: prevents results from being saved every five estimates.
- **quiet**: suppresses verbose output from the launcher.

Two methods can be applied on `TDistanceLikelihoodFunction` objects to change these options: `addCodeLauncherOpt` and `changeCodeLauncherOpt` (see *Specifying a Launcher object or options*).

```
# Adding more options to the code launcher
addCodeLauncherOpt(opt)
# Change the code launcher option
changeCodeLauncherOpt(opt)
```

The first method retains the default configuration and appends additional options, for example an option that enables distributed computation using the fork process (recall that this option is **"localhost=X"**, where X denotes the number of threads to use). In contrast, the second method resets the configuration entirely, allowing the user to define a new set of options from scratch.

Using the formalism introduced in *Recommended distance and likelihood functions, construction method*, a model instance based on the code `Foo`, initialised with `TCode`, can be created as shown below. This example assumes the model takes three inputs (**"ref_var1"**, **"par1"**, **"ref_var2"**) and it produces a single output, which is compared with the reference output **"ref_out1"** via the distance or likelihood function (see *Recommended distance and likelihood functions, construction method*).

```
# Define the dataservers
tdsRef = DataServer.TDataServer("reference", "myReferenceData")
# Load the data, both inputs (ref_var1 and ref_var2) and a single output (ref_out1).
tdsRef.fileDataRead("myInputData.dat")
...
tdsPar = DataServer.TDataServer("parameters", "myParameters")
tdsPar.addAttribute(DataServer.TNormalDistribution("par1", 0, 1))  # parameter to␣
↪calibrate
...
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
fout = Launcher.TOutputFileRow("_output_code_foo_.dat")
# The attribute in the output file
fout.addAttribute(DataServer.TAttribute("out"))
# Creation of the code
mycode = Launcher.TCode(tdsRef, "foo -s -k")
mycode.addOutputFile( fout )
...
# Create the instance of TCalClass
int ns=1
cal = Calibration.TCalClass(tdsPar, mycode, ns, "")
```

11.2.3.3 Construction for a function with the Launcher architecture

This constructor uses the **Launcher** architecture and is designed for C++ functions with the standard prototype.

In this case, the constructors have the following forms:

```
# Constructor with a function pointer using Launcher
TCalClass(tds, fcn, varexpinput, varexpoutput, ns=1, option="")
# Constructor with a function name using Launcher
TCalClass(tds, fcn, varexpinput, varexpoutput, ns=1, option="")
```

It takes up to six arguments, two of which are optional:

1. **tds**: a `TDataServer` object containing one attribute for each parameter to be calibrated. This is the `TDataServer` object called `tdsPar`, defined in *General introduction to data and model definition*.
2. **fcn**: the second argument is either the name of the function or a pointer to this function. The user must know the exact order of the input and output variables of this function.
3. **varexinput**: a colon-separated list of input variables, in the correct order, mixing reference attributes and parameter attributes.
4. **varexpoutput**: a colon-separated list of output variables.
5. **ns** (optional): the number of samples to be produced. This parameter is only relevant for methods that return multiple configurations (either multiple solutions to the minimisation problem or samples from the posterior distribution). It is not used for local minimisation with single-point initialisation, nor for linear Bayesian analysis (see *Analytical linear Bayesian estimation*). The **default** value is 1.
6. **option** (optional): the option that can be applied to the method. Options common to all calibration classes (those defined in the `TCalibration` class) are discussed in *Running the estimate*.

This is the simplest constructor: all information is provided in a single line, no extra options need to be defined, and no files must be created. Using the formalism introduced in *Recommended distance and likelihood functions, construction method*, the calibration object can be constructed using a function-pointer prototype of the code `Foo` as shown below. This example assumes the model takes three inputs (**"ref_var1"**, **"par1"**, **"ref_var2"**) and it produces a single output, which is compared with the reference output **"ref_out1"** via the distance or likelihood function (see *Recommended distance and likelihood functions, construction method*).

```
# Define the dataservers
tdsRef = DataServer.TDataServer("reference","myReferenceData")
# Load the data, both inputs (ref_var1 and ref_var2) and a single output (ref_out1).
tdsRef.fileDataRead("myInputData.dat")
...
tdsPar = DataServer.TDataServer("parameters","myParameters")
tdsPar.addAttribute(DataServer.TNormalDistribution("par1", 0, 1))  # parameter to␣
↪calibrate
...
# Create the instance of TCalClass
ns = 1
cal = Calibration.TCalClass(tdsPar, Foo, "ref_var1:par1:ref_var2", "out", ns, "")
```

11.2.3.4 Running the estimate

Once the calibration object is constructed and its distance or likelihood function created, the next step is to perform the calibration, that is to estimate the best value(s) or distribution of the parameters given the data. This is done by calling the method

```
estimateParameters(option="")
```

This method is generic and primarily calls an internal method, implemented in each class inheriting from `TCalibration`, where the actual estimate is performed.

There are different options that can be applied to the `estimateParameters` method, among which:

- **"saveAllEval"**: keeps all individual estimates in the internal dataserver, which can later be used to produce residuals plots (see *Drawing the residuals*). This may become impractical, as the number of stored estimates can be extremely large;

- **"noAgreement"**: this options removes the *agreement* attribute at the end of the estimate, if that information is deemed unnecessary.

Additional options, triggered via methods of the `TDistanceLikelihoodFunction`-inheriting instances, may also be useful for understanding and validating the calibration. For details, see *Available options for every distance and likelihood function*.

Several methods are available to represent the results. Their basic structure is defined in the `TCalibration` class, though some may not apply to specific methods. This section introduces the general framework while method-specific details are discussed in their respective sections.

11.2.3.5 Estimate custom residuals

Once the estimate has been performed, both *a priori* and *a posteriori* residuals can be computed. Since *a priori* and *a posteriori* estimates depend on the chosen algorithm, their formulas are not detailed here. It is also possible to re-evaluate residuals for any custom set of parameter values provided by the user. This can be done through

```
estimateCustomResiduals(resName, theta_nb, theta_val)
```

It takes three arguments, which are:

1. **resName**: a tag identifying the stored set of residuals;
2. **theta_nb**: the number of parameter values in the array (must match `_nPar`);
3. **theta_val**: an array containing the parameter values.

For convenience, this method can be called with a `numpy array`, as shown below

```
import numpy as np
mypar = np.array([0., 2., 3.])
mycal.estimateCustomResiduals("set1", len(mpar), mypar)
```

This feature allows re-estimating the residuals based on the available sample after the *a posteriori* distributions have been calibrated. A common use case is Markov chain Monte Carlo methods, where it may be necessary to discard the burn-in phase before evaluating residuals. Once created, the custom set can be used in the `drawResiduals` function, as discussed in *Drawing the residuals*.

11.2.3.6 Drawing the parameters

This method is used to visualize parameter values and posterior samples. The prototype is:

```
drawParameters(sTitre, variable = "*", option = "")
```

It takes up to three arguments, two of which are optional:

1. **sTitre**: the title of the plot (an empty string is allowed);
2. **variable** (optional): a list of parameter names to be drawn, separated by colons ":". The default **"*"** draws all parameters;
3. **option** (optional): a list of options, separated by commas "," to adjust the plotting behavior:
   - **"nonewcanvas"**: draw on the current canvas (instead of creating a new one);
   - **"vertical"**: if multiple parameters are plotted, display them stacked vertically (one per row). By default, plots are arranged horizontally, side by side.

Warning

If the `TMCMC` calibration object is used, the `drawParameters` method accounts for the potential *burn-in* and *lag* values by discarding the initial samples of the Markov chain and thinning it according to the specified values, in order to properly represent the behavior of the posterior distribution (see *Markov chain Monte Carlo approach*).

11.2.3.7 Drawing the residuals

This method is used to visualize the residuals of the output variables, i.e. the discrepancies between the model predictions and the reference outputs. Two types of residuals are typically considered:

- *A priori* residuals – based on the initial parameter values or the prior distribution;
- *A posteriori* residuals – obtained after the calibration procedure.

The prototype is:

```
drawResiduals(sTitre, variable = "*", select="1>0", option = "")
```

It takes up to four arguments, three of which are optional:

1. **sTitre**: the title of the plot (an empty string is allowed);
2. **variable** (optional): a list of parameter names for which one wants to draw the residuals, separated by colons ":". The default **"*"** draws all parameters;
3. **select** (optional): a selection expression that filters out configurations. For example, it can be used to exclude the *burn-in* period or to apply *lag* procedures in the MCMC algorithms (see *Markov chain Monte Carlo approach*);
4. **option** (optional): a list of options, separated by commas "," to adjust the plotting behavior:
   - **"nonewcanvas"**: draw on the current canvas (instead of creating a new one);
   - **"vertical"**: if multiple parameters are plotted, display them stacked vertically (one per row). By default, plots are arranged horizontally, side by side;
   - **"apriori/aposteriori"**: draw only the *a priori* residuals or only the *a posteriori* residuals. If neither option is specified, both are displayed;
   - **"custom=XXX"**: also draw custom residuals computed with `estimateCustomResiduals`.

For example, using the custom set "set1" defined earlier (see *Estimate custom residuals*), one can compare *a priori*, *a posteriori*, and custom residuals:

```
mycal.drawResiduals("Residuals title", "*", "", "nonewcanvas,apriori,aposteriori,
↪custom=set1")
```

Summary: `TCalibration` and `TDistanceLikelihoodFunction`

In summary, a calibration analysis should always follow these key steps:

1. Define both the parameter (`tdsPar`) and reference (`tdsRef`) objects, as explained in *General introduction to data and model definition*.
2. Define the model architecture (either **Relauncher** or **Launcher**-based) and the evaluator kind (code, C++-function, python-function…). Particular attention is needed when specifying the input and output attributes (the former should be an admixture of attributes from `tdsPar` and `tdsRef`, as explained in *General introduction to data and model definition*.

3. Define the `TCalibration`-inheriting object (meaning the underlying method chosen for this analysis) and construct it (see from *Construction with a TRun* to *Construction for a function with the Launcher architecture*).
4. Define the `TDistanceLikelihoodFunction`-inheriting object by calling `setDistance` or `setLikelihood` (depending on the `TCalibration` object), see the recommended distance and likelihood functions in *Recommended distance and likelihood functions, construction method*.
5. Set the method-dependent parameters (discussed in dedicated part, from *Minimisation techniques* to *Markov chain Monte Carlo approach*).
6. Call the `estimateParameters` method with selected options (see *Running the estimate*).
7. Perform the post-processing using method-dependent functions discussed in the dedicated sections, from *Minimisation techniques* to *Markov chain Monte Carlo approach*) and drawing methods (see also both *Drawing the parameters* and *Drawing the residuals*).

11.2.4 Use-case for this chapter

To illustrate the methods discussed in the following sections, in more detail than the previously introduced dummy examples, a general use-case will be used. This use-case relies on the `flowrate` model introduced (along with a descriptive sketch) in *The Launcher module* and whose equation is recalled below:

$$y = f(x) = \frac{2\pi T_u \left(H_u - H_l\right)}{\ln(\frac{r}{r_\omega}) \left[1 + \frac{2LT_u}{\ln(\frac{r}{r_\omega}) r_\omega^2 K_\omega} + \frac{T_u}{T_l}\right]}$$

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. $T_u \in [63\,070, 115\,600]$ $(m^2/year)$: Transmissivity of the superior layer of water;
4. $T_l \in [63.1, 116]$ $(m^2/year)$: Transmissivity 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 [Wor87]. For our purposes, the idea behind the upcoming examples (in this chapter, along with those in the use-case section, see *Macros Calibration*) is to consider that one has an observation sample. For this function, we consider that from all the inputs, only two have been varied (r_ω and L) and only one is actually unknown: H_l. The rest of the variables are set to fixed values: $r = 25050$, $T_u = 89335$, $T_l = 89.55$, $H_u = 1050$, $K_\omega = 10950$. 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 /␣
```

(continues on next page)

(continued from previous page)

```
↪tl);

    y[0] = num / den;
}
```

As discussed previously, the function assumes that the user is aware of the input order. 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
ExpData = "Ex2DoE_n100_sd1.75.dat"

# define the reference
tdsRef = DataServer.TDataServer("tdsRef", "doe_exp_Re_Pr")
tdsRef.fileDataRead(ExpData)

# define the parameters
tdsPar = DataServer.TDataServer("tdsPar", "tdsPar")
tdsPar.addAttribute(DataServer.TAttribute("hl", 700.0, 760.0))  # if stochastic laws␣
↪are needed
# use tdsPar.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 760.0))

# Create the output attribute
out = DataServer.TAttribute("out")
```

11.3 Minimisation techniques

Warning

This method is entirely reliant on the **Relauncher** architecture so the only available constructor is the runner constructor (discussed in the section *Construction with a TRun*). This means there is no constructor based on `TCode` or function (respectively described in *Construction with a TCode* and *Construction for a function with the Launcher architecture*). This is because the method uses the **Nlopt**-algorithm library, introduced in *The Reoptimizer module* or the **Vizir** package for multi-criteria and many-objective algorithms.

Even though the theory behind this method is not revolutionary, these methods are of interest and are, historically and conceptually, among the simplest methods one can use. Because of the way the framework is organised, it can be used with all **Relauncher** assessors and can call any algorithm from **NLopt** or **Vizir**.

Apart from what is presented in *Defining the TMinimisation properties*, the `TMinimisation` class does not provide any additional options or methods beyond those available in the default `TCalibration` object (see *Calibration classes, distance and likelihood functions, observations and model*).

The usage of the `TMinimisation` class can be summarised in a few key steps:

1. Prepare the data and the model:
    - Specify the experimental dataserver, the parameters to calibrate, and the model;
    - Construct the `TMinimisation` object with the appropriate distance function (see *Constructing the TMinimisation object*).
2. Set the algorithm properties:

- Define optimisation algorithm and prepare its properties (see *Defining the TMinimisation properties*).

3. Perform the estimate and analyse the results:
   - Run the estimate process;
   - Extract the results and visualise them with the standard plotting tools (see *Looking at the results*).

11.3.1 Constructing the `TMinimisation` object

As stated above, the only available constructor is the one whose prototype includes an instance of a **TRun**-inheriting object. This approach provides a simple way to change the evaluator (e.g., from a C++ function to a Python function or an external code) and to use either a sequential approach (for a code) or a threaded one (to distribute locally the estimates).

In this case, the constructor should look like this:

```
# Constructor with a runner
TMinimisation(tds, runner, ns=1, option="")
```

This method takes up to four arguments, two of which are optional:

1. **tds**: a `TDataServer` object containing one attribute for each parameter to be calibrated. This is the `TDataServer` object called `tdsPar`, defined in *General introduction to data and model definition*.
2. **runner**: a `TRun`-inheriting instance that contains all the model information and whose type determines how the estimates are distributed: it can either be a `TSequentialRun` instance or `TThreadedRun` for distributed computations.
3. **ns** (optional): the number of samples to be produced. This parameter is not used in this context.
4. **option** (optional): the options that can be applied to the method. Options common to all calibration classes (those defined in the `TCalibration` class) are discussed in *Running the estimate*.

A crucial step in this constructor is the creation of the instance that inherits from `TRun`. As mentioned earlier, its type determines how the estimates are distributed. To construct such an object, one needs to provide an evaluator (the available evaluators are described in detail in *TEval*).

The final step is to construct the `TDistanceLikelihoodFunction`, a mandatory step that must always follow the constructor, using the `setDistance` method (following the prototype presented in *Recommended distance and likelihood functions, construction method*). Advanced users also have the option to define a custom distance by following the prototype:

```
setDistance(distFunc, tdsRef, input, reference, weight="")
```

11.3.2 Defining the `TMinimisation` properties

Once the `TMinimisation` instance is created along with its associated `TDistanceLikelihoodFunction`, the optimisation properties must be defined to specify which algorithm will be used. This is done by calling the `setOptimProperties`, whose prototype is:

```
# Prototype for NLopt or Vizir
setOptimProperties(solv, option="")
```

The option field is reserved for future use and is currently non-functional. It is advisable to prepare the solver before integrating it into the `TCalibration` object, as modifying it afterwards may be difficult (as shown in the example below). For example, it may be preferable to set the population size, number of iterations, and number of steps using the `setSize` method on the `TVizirGenetic` object before to provide it to `TMinimisation` (via `setOptimProperties`), which will then automatically create the appropriate optimisation master based on the solver type:

- **TNlopt**: the optimisation master used for any `TNloptSolver` instance. An example can be found in *Macro "calibrationMinimisationFlowrate1D.py"*;
- **TVizir2**: the optimisation master used for any `TVizirSolverShare` instance. Even though the optimisation will remain mono-criterion, this might be useful for certain complex problems, as discussed in *Macro "calibrationMinimisationFlowrate2DVizir.py"*.

It is also possible to access the optimisation master by calling the `getOptimMaster` method. This method returns a pointer to the newly created `TOptimShare` instance. This might be useful to set some properties (for instance the tolerance).

```
# Set the calibration object
cal = Calibration.TMinimisation(tdsPar,runner,1)
cal.setDistance("relativeLS",tdsRef,"rw:1","Qexp")

# Set Vizir optimisation properties
solv = Reoptimizer.TVizirGenetic()
solv.setSize(24,15000,100)
cal.setOptimProperties(solv)
optimMaster = cal.getOptimMaster()
optimMaster.setTolerance(1e-6)

# Set Nlopt optimisation properties
solv = Reoptimizer.TNloptSubplexe()
cal.setOptimProperties(solv)
optimMaster = cal.getOptimMaster()
optimMaster.setTolerance(1e-6)
```

11.3.3 Looking at the results

Finally, once the computation is complete (using the standard `estimateParameters` method), the calibrated values of the parameters are directly stored in the parameter `TDataServer`.

It is also recommended to check the residuals (using the standard `drawResiduals` method, introduced in *Drawing the residuals*), as they may provide useful information on the quality of the calibration.

No additional options or visualization methods are specific to the `TMinimisation` implementation; these functions behave exactly as described in their respective sections.

Two examples are also provided in the use-case section (see *Macro "calibrationMinimisationFlowrate1D.py"* and *Macro "calibrationMinimisationFlowrate2DVizir.py"*).

11.4 Analytical linear Bayesian estimation

This method is fairly simple from an algorithm point of view as it consists mainly of the analytical formulation of the posterior distribution under assumptions: the problem can be considered linear and the prior distributions are normally distributed (or non-informative/flat, as discussed in [Bla17]).

In practice, this technique is applied by following the procedure provided in *Calibration classes, distance and likelihood functions, observations and model* with one important difference, however: the code or function passed through the constructor of the `TLinearBayesian` object is not strictly necessary. The parameter estimate is analytical so the main point of providing an assessor is to get both the *a priori* and *a posteriori* residuals distributions.

The usage of the `TLinearBayesian` class can be summarised in a few key steps:

1. Prepare the data and the model:

- Select the assessor type to be used and construct the `TLinearBayesian` object with the appropriate likelihood function via `setLikelihood` method (see *Constructing the TLinearBayesian object*).

2. Set the algorithm properties:
   - Provide the input covariance matrix, *i.e.*, the reference observation covariance (in [Bla17], this corresponds to Σ). This step is mandatory, as the covariance matrix is used to compute the posterior distribution, as discussed in [Bla17];
   - Specify 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 several input variables. This is discussed in *Defining the TLinearBayesian properties*;
   - A transformation function may be provided, although this is optional. This is discussed in *Transformation of the results*.
3. Perform the estimate and analyse the results:
   - Run the estimate process;
   - Extract the results and visualise them with the standard plotting tools (see *Looking at the results*).

11.4.1 Constructing the `TLinearBayesian` object

The constructors available for creating an instance of the `TLinearBayesian` class are detailed in *Common methods of the calibration classes*. As a reminder, the available prototypes are:

```
# Constructor with a runner
TLinearBayesian(tds, runner, ns=1, option="")
# Constructor with a TCode
TLinearBayesian(tds, code, ns=1, option="")
# Constructor with a function using Launcher
TLinearBayesian(tds, fcn, varexpinput, varexpoutput, ns=1, option="")
```

Details about these constructors can be found in *Construction with a TRun*, *Construction with a TCode*, and *Construction for a function with the Launcher architecture* respectively for the `TRun`, `TCode`, and `TLauncherFunction`-based constructor. In all cases, the number of samples ns is set to 1 by default, and modifying it has no effect on the results, since it returns the analytical distributions. This class does not define any specific options.

The final step is to construct the `TDistanceLikelihoodFunction`, a mandatory step that must always immediately follow the constructor. This can be done via the `setLikelihood` method (following the prototype presented in *Recommended distance and likelihood functions, construction method*).

Warning

The Analytical Linear Bayesian Estimation method relies on the assumption that residuals are Gaussian (centered) with a defined covariance matrix (see *Defining the TLinearBayesian properties*). As a result, the likelihood is not a free choice but is determined by these assumptions.

It is important to note that the `setLikelihood` method is used only for residual calculation, to compare the prior and posterior. Regardless of the initial choice of likelihood, the function is locally redefined so that the computation is performed via the `lin_gauss` function using matrix multiplication (through the `TMahalanobisDistance` method).

Although not recommended, it is still possible to define a custom likelihood function:

```
setLikelihood(likelihoodFunc, tdsRef, input, reference, weight="")
```

11.4.2 Defining the `TLinearBayesian` properties

Once the `TLinearBayesian` instance is created along with its `TDistanceLikelihoodFunction`, two methods must be called before performing parameter estimate. These methods are mandatory, as they define the analytical formula used to obtain the Gaussian parameter values of the *a posteriori* distribution (see [Bla17]).

The first method (although the order is not important) is `setRegressorName`, whose prototype is

```
setRegressorName(regressorname)
```

The only argument is `regressorname`, a string containing the list of regressor names separated by ":". The method then performs two checks: it verifies that the number of regressors matches the number of parameters to be calibrated and it checks that every regressor name provided matches one existing attribute in the reference `TDataServer` (`tdsref`). If the observation `TDataServer` does not contain the regressors (when the input file is loaded) these attributes must be constructed from scratch, either with `TAttributeFormula` or by using another dedicated assessor (as done in the use-case shown in *Macro "calibrationLinBayesFlowrate1D.py"*).

The other method is `setObservationCovarianceMatrix`, whose prototype is

```
setObservationCovarianceMatrix(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 dimensions (both rows and columns must be equal to n);
- the provided matrix should be symmetrical;

Once these conditions are satisfied, estimate can proceed. One can find an example of how to use these methods in the use-case dedicated subsection (see *Macro "calibrationLinBayesFlowrate1D.py"*).

11.4.3 Looking at the results

Finally, once the computation is complete there are three different kinds of results, with several ways to interpret and transform them. This section describes some important and specific aspects of these results.

11.4.3.1 Transformation of the results

The idea is that when you want to consider your model as linear, you may need to slightly transform it to ensure proper linear behaviour and to express and compute the needed regressors. For this particular situation, the use-case described in *Macro "calibrationLinBayesFlowrate1D.py"* will be used. In this case, the `flowrate` function should be linearised as follows:

$$f_\theta(x) = (2\pi T_u)\left(\ln(\frac{r}{r_\omega})\left[1 + \frac{2LT_u}{\ln(\frac{r}{r_\omega})r_\omega^2 K_\omega} + \frac{T_u}{T_l}\right]\right)^{-1}\theta = H \times \theta$$

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 this, it is clear that we will be calibrating a newly defined parameter $\theta = (H_u - H_l)$. Therefore, at some point, we will need to transform it back into our parameter of interest.

This is the reason why the `setParameterTransformationFunction` method has been implemented: to transform the estimated parameters given the linear regressor, the observation covariance matrix, and the prior distribution. Since the transformations, if they exist (they are optional), are expected to be carried out using simple operations with constant values, they should affect only the mean vector and not the covariance matrix of the posterior multivariate normal distribution. The prototype of this function is as follows:

```
setParameterTransformationFunction(fTransfoParam)
```

Its only argument is a pointer to the transformation function. This function, used to obtain the transformed parameter values, takes two arguments: the input parameters, which are the raw values estimated from the analytical formula detailed in [Bla17], and the output parameters, which correspond to the desired transformed values. Both arguments are double arrays of length equal to the number of parameters.

The example provided in the use-case *Macro "calibrationLinBayesFlowrate1D.py"* is simple as there is only one parameter to be estimated, which implies that both arguments are one-dimensional double arrays which should look like this:

```
void transf(double *x, double *res)
{
    res[0] = 1050 - x[0];  // simply H_l = \theta - H_u
}
```

Warning

To achieve this in Python, the function must be placed in a C file (for example, `myFunction.C`), and this file must be loaded to obtain a handle on the function. The following lines summarize these two steps in a fictional macro that defines 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)
```

11.4.3.2 Accessing the results

Once the estimate is done (via the classic `estimateParameters` method), the results can be accessed by calling three methods, detailed below. All three functions share the same prototype: they take no arguments and return a `TMatrixD` instance containing the corresponding information. The functions are:

- **getParameterValueMatrix**: it returns a `TMatrixD` with one raw filled with the calibrated values of the parameters calculated from the analytical formula (one-dimensional);
- **getParameterCovarianceMatrix**: it returns a `TMatrixD` filled with the covariance matrix of the estimated parameters (symmetric and $(nPar, nPar)$-dimensional);
- **getTransfParameterValueMatrix**: it returns a `TMatrixD` with one raw filled with the transformed values of the parameters, in case `setParameterTransformationFunction` has been correctly applied (one-dimensional).

11.4.3.3 Drawing the parameters

Parameters can be plotted using a newly defined `drawParameters` instance, which shares the same prototype as the original method described in *Drawing the parameters*.

```
drawParameters(sTitre, variable = "*", option = "")
```

It takes up to three arguments, two of which are optional:

1. **sTitre**: the title of the plot (an empty string is allowed);

2. **variable** (optional): a list of parameter names to be drawn, separated by colons ":". The default **"*"** draws all parameters;
3. **option** (optional): a list of options, separated by commas "," to adjust the plotting behavior:
    - **"nonewcanvas"**: draw on the current canvas (instead of creating a new one);
    - **"vertical"**: if multiple parameters are plotted, display them stacked vertically (one per row). By default, plots are arranged horizontally, side by side.
    - **"apriori/aposteriori"**: draw only the *a priori* residuals or only the *a posteriori* residuals. If neither option is specified, both are displayed;
    - **"transformed"**: this option specifies that the transformed values should be used as the mean vector of the multivariate normal posterior distribution.

The main difference compared to the standard `drawParameters` method in `TCalibration` is that the plotted object are analytical functions.

In addition to the parameters, the residuals can be plotted using the standard `drawResiduals` method, which remains unchanged (see *Drawing the residuals*).

11.4.4 Prediction of the variance

Once the estimate is completed, it is possible to compute the central value for a new set of input values (i.e., for a new design-of-experiments) using the newly estimated parameter values. Although this applies to every method in the calibration module, the Linear Bayesian procedure has the advantage of providing the covariance matrix of the parameters. Under some assumptions on the input distribution, it is possible to obtain a variance for each new predicted value, reflecting only the uncertainty due to the parameters. For more details on the estimate, see [Bla17]. This can be done by calling the `computePredictionVariance` method:

```
computePredictionVariance(tdsPred, outname)
```

This method takes two arguments, which are:

1. **tdsPred**: a `TDataServer` containing the new locations to be estimated, in which **all** regressors must be available in order to be able to compute the covariance matrix;
2. **outname**: the name of the attribute to be created, which will be filled with the diagonal elements (the variances) of the Σ_θ^{pred} matrix.

An example is also provided in the use-case section (see *Macro "calibrationLinBayesFlowrate1D.py"*).

11.5 Approximate Bayesian Computation techniques (ABC)

This section covers methods grouped under the acronym **ABC**, which stands for *Approximate Bayesian Computation*. The core idea is to perform Bayesian inference without explicitly evaluating the model likelihood function. For this reason, these methods are also refered to as **likelihood-free** algorithms [Wil13].

As a reminder, the principle of the Bayesian approach is summarized 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 $L(\mathbf{y}|\theta)$ is the conditional probability of the observations given the parameter values 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 [Bla17].

From a technical perspective, the methods in this section inherit from the `TABC` class (which itself inherits from `TCalibration`, in order to benefit from all standard features). Currently, the only implemented ABC method is the Rejection algorithm, presented in [Bla17], whose implementation is provided through the `TRejectionABC` class described below.

The usage of the `TRejectionABC` class can be summarised in a few key steps:

1. Prepare the data and the model:
    - The parameters to be calibrated must be instances of classes inheriting from `TStochasticAttribute`;
    - Select the assessor type and construct the `TRejectionABC` object with the appropriate distance function (see *Constructing the TRejectionABC object*).
2. Set the algorithm properties:
    - Define optional behaviours;
    - Specify the uncertainty hypotheses via the dedicated methods (see *Defining the TRejectionABC properties*).
3. Perform the estimate and analyse the results:
    - Run the estimate process;
    - Extract the results and visualise them with the standard plotting tools (see *Looking at the results*).

11.5.1 Constructing the `TRejectionABC` object

The constructors available for creating an instance of the `TRejectionABC` class are detailed in *Common methods of the calibration classes*. As a reminder, the available prototypes are:

```
# Constructor with a runner
TRejectionABC(tds, runner, ns=1, option="")
# Constructor with a TCode
TRejectionABC(tds, code, ns=1, option="")
# Constructor with a function using Launcher
TRejectionABC(tds, fcn, varexpinput, varexpoutput, ns=1, option="")
```

Details about these constructors can be found in *Construction with a TRun*, *Construction with a TCode*, and *Construction for a function with the Launcher architecture* respectively for the `TRun`, `TCode`, and `TLauncherFunction`-based constructor. In all cases, the number of samples ns must be specified, as it represents the number of retained samples in the final posterior distribution. In our implementation, the total number of computations is determined by the chosen percentile value (see *Defining the percentile*).

This class provides one specific option, which can be used to modify the default value of the *a posteriori* distribution returned by the algorithm. Two possible choices are available for obtaining the single-point estimate that best represents the distribution:

- Mean of the distribution: this is the **default** option;
- Mode of the distribution: the user must specify **"mode"** in the option field of the `TRejectionABC` constructor.

The default solution is straightforward, whereas the second requires internal smoothing of the distribution in order to obtain the best estimate of the mode.

The final step is to construct the `TDistanceLikelihoodFunction`, a mandatory step that must always immediately follow the constructor. Although ABC methods are Bayesian, they are likelihood-free: instead of directly computing the likelihood, they calculate the distance between the observed data and the simulated data for several prior samples, until the best-fitting distribution is identified. Consequently, the `TDistanceLikelihoodFunction` object can be constructed via the `setDistance` method (following the prototype presented in *Recommended distance and likelihood functions, construction method*). Advanced users also have the option to define a custom distance by following the prototype:

```
setDistance(distFunc, tdsRef, input, reference, weight="")
```

> **Warning**
>
> If the reference dataset is compared against a deterministic model (i.e. a model with no intrinsic stochasticity), it is necessary to explicitly specify the uncertainty hypotheses. This is done via the method described in *Introducing noise for deterministic function*.

11.5.2 Defining the `TRejectionABC` properties

Once the `TRejectionABC` instance is created along with its associated `TDistanceLikelihoodFunction`, a few additional methods can be used to fine-tune the algorithm parameters. These methods are optional, since each property has a predefined default value. The available options are described in the following sub-sections.

11.5.2.1 Defining the percentile

The first method discussed here is straightforward: the principle of rejection is to keep the best-tested configurations. This can be done either by applying a threshold value on the distance results (called δ in [Bla17]) or by retaining a fixed fraction of the tested configurations, defined through a percentile ε_{Dist}. The `TRejectionABC` method implements the latter approach. By **default**, the percentile is set to $\varepsilon_{Dist} = 0.01$.

To modify this value, the user can call `setPercentile`, whose prototype is

```
setPercentile(eps)
```

where the argument `eps` specifies the fraction of configurations to be kept.

An important consequence is that the total number of configurations evaluated is computed as follows:

$$n_{\text{Comp}} = \frac{n_S}{\varepsilon_{Dist}}$$

where n_S is the number of retained samples in the final posterior distribution, as defined in the constructor (see *Constructing the TRejectionABC object*).

11.5.2.2 Introducing noise for deterministic function

As previously explained, when comparing your reference dataset to a deterministic model (i.e., a model with no intrinsic stochastic behaviour), the user can explicitly specify his own uncertainty assumptions. This can be done by calling `setGaussianNoise`, whose prototype is

```
setGaussianNoise(stdname)
```

The idea is to inject random noise (assumed Gaussian and centered) into the model predictions, using internal variables from the reference dataset to define its standard deviation. The only argument is a list of variables, formatted as `"stdvaria1:stdvaria2"`, where each element corresponds to a variable within the reference `TDataServer`. These values provide the standard deviation for each observation point (for example, to represent experimental uncertainty).

This solution allows to:

- define a common uncertainty (applied generally across all observations in the reference dataset) by simply adding an attribute with a `TAttributeFormula`, where the formula is constant;
- use experimental uncertainties that are provided along with the reference values;
- store all hypotheses within the reference `TDataServer` object. For this reason, we strongly recommend saving both the parameter and reference datasets at the end of a calibration procedure.

> **Warning**
>
> The number of variables in the `stdname` list must match the number of model outputs. Even in the special case where calibration involves two outputs, with one having no associated uncertainty, a zero-valued attribute should still be added for that output if the other requires an uncertainty model.

11.5.3 Looking at the results

Finally, once the computation is complete (using the standard `estimateParameters` method), the results can be examined in two main ways (in addition to inspecting the final datasets):

- using the `drawParameters` method, introduced in *Drawing the parameters*;
- using the `drawResiduals` method, introduced in *Drawing the residuals*.

No additional options or visualization methods are specific to the `TRejectionABC` implementation; these functions behave exactly as described in their respective sections.

An example is also provided in the use-case section (see *Macro "calibrationRejectionABCFlowrate1D.py"*).

11.6 Markov chain Monte Carlo approach

In a Bayesian framework, **Markov Chain Monte Carlo** (MCMC) methods are a powerful tool for calibration. They are especially valuable when the statistical model cannot be solved analytically, such as when the prior distribution has a complex structure or the model is nonlinear. Unlike many classical approaches, MCMC does not require the assumption of Gaussian errors: it remains applicable even when the likelihood is non-Gaussian.

Rather than providing a single "best-fit" solution (as in minimisation techniques), MCMC generates a collection of parameter samples that represent the full posterior distribution (similar to ABC methods). However, these methods also come at the cost of potentially high computational demand, since long sampling chains may be required to achieve convergence and reliable estimates. Users should therefore interpret results as distributions and ensure that convergence diagnostics are checked before drawing conclusions (see [Bla17] for more details).

The usage of the `TMCMC` class can be summarised in a few key steps:

1. Prepare the data and the model:
   - The parameters to be calibrated must be instances of classes inheriting from `TStochasticAttribute`;
   - Select the assessor type and construct the `TMCMC` object with the appropriate likelihood function (see *Constructing the TMCMC object*).
2. Set the algorithm properties:
   - Define optional behaviours;
   - Specify the uncertainty hypotheses via the dedicated methods (see *Defining the TMCMC properties*).
3. Perform the estimate:
   - Run the estimate process;
   - Eventually continue it if the convergence is not reached (see *Running the estimate, exporting and loading chains, and continuing the calculation*).
4. Perform post-processing:
   - Investigate the quality of the samples through diagnostics and plots (see *Investigating the quality of the samples through diagnostics and plots*).
5. Analyse the results:

- Extract the results and visualise them with the standard plotting tools (see *Looking at the results*).

11.6.1 Constructing the TMCMC object

The constructors available for creating an instance of the TMCMC class are detailed in *Common methods of the calibration classes*. As a reminder, the available prototypes are:

```
# Constructor with a runner
TMCMC(tds, runner, ns=1, option="")
# Constructor with a TCode
TMCMC(tds, code, ns=1, option="")
# Constructor with a function using Launcher
TMCMC(tds, fcn, varexpinput, varexpoutput, ns=1, option="")
```

Details about these constructors can be found in *Construction with a TRun*, *Construction with a TCode*, and *Construction for a function with the Launcher architecture* respectively for the TRun, TCode, and TLauncherFunction-based constructor. In all cases, the number of iterations ns must be specified.

This class provides one specific option, which can be used to modify the default value of the *a posteriori* distribution returned by the algorithm. Two possible choices are available for obtaining the single-point estimate that best represents the distribution:

- Mean of the distribution: this is the **default** option;
- Mode of the distribution: the user must specify **"mode"** in the option field of the TMCMC constructor.

The default solution is straightforward, whereas the second requires internal smoothing of the distribution in order to obtain the best estimate of the mode.

In practice, the constructor—whichever one is chosen—initializes a folder named MCMC_N, where N is an integer ensuring the folder name is available. This folder contains all the information related to the MCMC calculation (the chain values, the algorithm used, the number of accepted samples, etc.), with each chain stored in a separate file named MCMC_N_chain_M (for the M-th chain, starting from 0). By default, only one chain is initialized, so the MCMC_N folder contains a single file MCMC_N_chain_0. This folder will be duplicated once the setMultistart method is called (see *Initialising several chains*). The files are automatically loaded and saved during the computation.

The next step is to construct the TDistanceLikelihoodFunction, a mandatory step that must always immediately follow the constructor. For now, the only available likelihood function is the Gaussian, as it is the most commonly used. It can be accessed through the classic setLikelihood method using the function name "log-gauss", following the standard prototype (presented in *Recommended distance and likelihood functions, construction method*). As previously explained, in this case the weights provided to the constructor correspond to the standard deviations of each observation. Advanced users also have the option to define a custom likelihood by following the prototype:

```
setLikelihood(likelihoodFunc, tdsRef, input, reference, weight="")
```

11.6.2 Defining the TMCMC properties

Once the TMCMC instance is created along with its TDistanceLikelihoodFunction, a few methods are available to tune the algorithms. All of these methods are optional, as default values are provided. However, they should be applied in the following order:

1. Define the MCMC algorithm properties that are shared by all chains: MCMC algorithm, acceptation range, and information display interval (these can be set in any order).
2. Specify the number of chains to initialize.
3. Prepare the starting points and the initial standard deviations of the proposal distribution (these can be set in any order).

11.6.2.1 Chosing the MCMC algorithm

Several MCMC algorithms are available, and the desired algorithm can be selected using the `setAlgo` method, which has the following prototype:

```
setAlgo(algoMCMC)
```

The method takes a single string argument that specifies the algorithm to use. Currently, the following options are supported:

- Component-wise Metropolis–Hastings, specified with **"MH_1D"**. This method updates one parameter at a time, which is useful when parameters have different influences on the model. By generating new candidates along one direction at a time, it prevents less influential parameters from being overshadowed by the more influential ones. This approach is generally slower than the classic Metropolis–Hastings;
- Classic Metropolis–Hastings, specified with **"MH_multiD"** (default algorithm). This method generates candidates in all directions simultaneously, making it faster than the component-wise approach, but it may be less efficient when parameter influences vary widely.

> **Warning**
>
> As explained, this method should be done immediately after creating the `TMCMC` object and assigning its likelihood.

11.6.2.2 Tuning the acceptation rate

As also explained in [Bla17], there are theoretical discussions on the acceptation rate expected, depending on the dimension of the parameter space for instance. As stated in some references (see [GRG+96, RGG+97]) when well initialised, for a single-dimension problem, the acceptation rate could be around 44%, decreasing to about 23% as the dimensionality increases. By **default**, the exploration parameters are set during the initialisation step (see *Initialising the process*).

If one wants to change this, a possible approach is to use the `setAcceptationRatioRange` method whose prototype is the following

```
setAcceptationRatioRange(lower, higher)
```

This prototype takes two arguments: the lower (r_{low}) and higher (r_{hig}) bounds. The idea is simply that after a certain number of estimate (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 possible (when the acceptation rate is called r_{acc}):

- $r_{\text{acc}} \leq r_{\text{low}}$: the acceptation rate is too low compared to the desired range, this means the algorithm is moving too far from the last accepted configuration. As a consequence, the variation range is reduced by 10% to improve convergence.
- $r_{\text{hig}} \leq r_{\text{acc}}$: the acceptation rate is too high compared to the desired range, this means that the algorithm is not exploring the parameter space adequately, focusing only on a well-performing region. As a consequence, the variation range is increased by 10% to encourage exploration.
- $r_{\text{low}} \leq r_{\text{acc}} \leq r_{\text{hig}}$: the acceptance rate is within the desired range, no adjustment is made.

When the `setAcceptationRatioRange` method has been called, this process will be called at the end of every batch. Otherwise, the **default** behaviour is to leave the acceptation rate unchanged, as no boundaries have been set.

From the prototype and the behaviour discussed above, two rules must be followed when using this method: $(r_{\text{low}}, r_{\text{hig}}) \in [0, 1]^2$ and $r_{\text{low}} < r_{\text{hig}}$.

Finally, the evolution of the acceptance rate can be examined by plotting the relevant information, as discussed in *Drawing the acceptation ratio*.

11.6.2.3 Information on the process

The next method discussed here is fairly simple: a MCMC simulation may take a long time to complete, particularly when running code, regardless of the chosen architecture. This method changes the number of iterations between two output displays. The prototype is simply:

```
setNbDump(nbDump)
```

The method takes a single integer argument that specifies the interval (modulo) at which the algorithm displays its progress. With the **default** value of 1000, the output will look like this:

```
1000 events done
2000 events done
3000 events done
...
```

11.6.2.4 Initialising several chains

As already explained in [Bla17], initialising multiple chains can help assess the convergence of MCMC algorithms more reliably. Since these algorithms are inherently sequential (each state of the chain depending on the previous one), running several chains in parallel also makes it possible to exploit modern computational power more effectively. At present, this parallelization is not yet implemented—the chains are still computed sequentially—but it will be supported in a future release.

If you wish to initialise multiple chains, this should be done using the following prototype:

```
setMultistart(nb_multistart)
```

The method takes a single integer argument that specifies the number of chains to be initialised.

> **Warning**
>
> As explained, this method duplicates the file of the initial chain to prepare several chains, preserving the properties that have already been defined. For this reason, it should be performed immediately after selecting the MCMC algorithm, to ensure that all chains use the same algorithm. If the default algorithm is used, this step should be done after creating the `TMCMC` object and assigning its likelihood.

11.6.2.5 Initialising the process

As already explained in [Bla17], the MCMC algorithm starts at a given point and then moves to a new configuration. This means that two things must be specified for each parameter: a starting value, and a variation range used to move to the next location. Since all the input parameters are stochastic (in other words, instances of classes inheriting from `TStochasticAttribute`), the **default** values are

- randomly drawn for the starting point;
- the theoretical standard deviation used as the variation range.

The default behaviour can easily be overridden by calling the `setStartingPoints` and the `setProposalStd` methods with these prototypes:

```
# Prototype setStartingPoints
setStartingPoints(ichain, values)
# Prototype setProposalStd
setProposalStd(ichain, standDev)
```

The first argument of both prototypes is the chain index, an integer between -1 and $n_C - 1$, where n_C is the number of chains (indexed from 0 to $n_C - 1$). If -1 is chosen, all chains will be assigned the same starting point (not recommended, since the purpose of using multiple chains is to initialise them at different locations to verify convergence toward the same region) or the same standard deviation.

The second argument of both prototypes is a vector containing either the initial values of the starting points (one value for each parameter to calibrate) or the standard deviations (one for each parameter), which will be assigned to the corresponding chain. This means that the vectors must have a size p, corresponding to the number of parameters.

11.6.3 Running the estimate, exporting and loading chains, and continuing the calculation

The computation can be performed using the standard `estimateParameters` method, which automatically saves the chains in their associated files. Although it is generally unnecessary (since saving is done automatically), it is still possible to manually export the results of a chain or reload them later using `export_chain_MCMC` and `read_chain_MCMC`, whose prototypes are the following:

```
# Export one chain
export_chain_MCMC(fileName)
# Read one chain
read_chain_MCMC(fileName)
```

Both methods take a single string argument specifying the file name used for exporting or loading the chain results.

In most cases, these methods are not required. They become useful when a computation finishes without reaching convergence and the user wishes to extend the run. This can be done with the `continueCalculation` method, which executes additional iterations of the MCMC algorithm in order to try to achieve convergence. Its prototype is:

```
continueCalculation(new_Ns)
```

This method takes a single integer argument that specifies the number of additional iterations to run. In this case, it may be useful to reload one of the previously computed chains with `read_chain_MCMC` before calling `continueCalculation`. Regardless of which chain is loaded, the calculation will continue for all initialized chains.

11.6.4 Investigating the quality of the samples through diagnostics and plots

Computing iterations does not guarantee that the chains have reached convergence and are properly sampling the posterior distribution. Although there is no exact way to prove convergence, several techniques can help assess the quality of the samples. Before plotting and analyzing the results, it is mandatory to check these diagnostics to determine whether the results are reliable or if the algorithm should be run for additional iterations (see *Running the estimate, exporting and loading chains, and continuing the calculation*). These techniques serve two main purposes:

Ensuring convergence of the chains: Convergence means that the chains have stabilized around the same region. This must be checked by the user using the trace plot (see *Drawing the trace*). It is also recommended to check the stability of the acceptance ratio, which should typically lie between 20% and 50%, using the acceptance ratio plot (see *Drawing the acceptation ratio*). Finally, the user must define the **burn-in** (also called warm-up), i.e., the number of initial iterations discarded before the chain stabilizes. This is done with the `setBurnin` method:

```
setBurnin(burnin)
```

This method takes a single integer argument that specifies the number of iterations to remove (non-converged iterations).

When multiple chains are initialized (at least 4), it is also possible to compute the **Gelman–Rubin** statistic, which compares intra- and inter-chain variances (see *Checking for convergence with the Gelman–Rubin diagnostic*). Values close to 1 indicate good convergence.

Ensuring approximate independence of posterior samples: Because Markov chains generate dependent samples, there is a risk that successive samples are correlated and do not explore the posterior distribution effectively. To assess this, the **Effective Sample Size (ESS)** can be computed with the `diagESS` method (see *Thinning the chains with ESS*), which provides the equivalent number of independent samples. This method also suggests an appropriate **lag value**, i.e., the number of iterations to skip before selecting the next uncorrelated sample. The lag is set using the `setLag` method:

```
setLag(lag)
```

The method takes a single integer argument that specifies the lag value.

Warning

Setting lag and burn-in values with the `setLag` and `setBurnin` methods will affect most drawing methods, and a line will indicate which default cut and lag were applied to produce the plot. The only exception is the residuals plot, since the *a posteriori* residuals are computed during `estimateParameters`, before any burn-in or lag is applied. To re-estimate residuals with a specific lag or burn-in, use the `estimateCustomResiduals` method (see *Estimate custom residuals*). To remove the current cut and lag, use the `clearDefaultCut` method, which requires no arguments and clears both lag and burn-in:

```
clearDefaultCut()
```

Another important point is that these methods may discard a significant number of samples. The burn-in length and lag value should therefore be chosen carefully to balance two goals: keeping enough points to ensure a good representation of the posterior distribution, while also guaranteeing convergence and independence of the samples.

11.6.4.1 Drawing the trace

This method is used to visualise the trace of the chains, i.e., the evolution of parameter values across iterations. It should be used to check the stability of the chains (stability of the mean and variance, see [Bla17]), and to detect potential autocorrelation (when the chain moves too slowly within the target distribution).

Thus, trace plots provide a first indication of the appropriate **burn-in** (the number of iterations to discard before stability is reached) and **lag** (the number of iterations to skip to reduce autocorrelation). The prototype is:

```
drawTrace(sTitre, variable = "*", option = "")
```

This method takes up to three arguments, two of which are optional:

1. **sTitre**: the title of the plot (an empty string is allowed);
2. **variable** (optional): a list of parameter names to be drawn, separated by colons ":". The default `"*"` draws all parameters;
3. **option** (optional): a list of options, separated by commas ",", to adjust the plotting behavior:
   - **"nonewcanvas"**: draw on the current canvas (instead of creating a new one);
   - **"vertical"**: if multiple parameters are plotted, display them stacked vertically (one per row). By default, plots are arranged horizontally, side by side;

Returning to the example in *Macro "calibrationMCMCFlowrate1D.py"*, the trace plots (shown in Figure 13.66) reveal that the very beginning of the chain is unstable, likely because initialization was far from the most probable value. Afterward,

the behavior stabilizes, oscillating around what appears to be the most probable region. Based on this, a small burn-in of about 10 iterations could be chosen.

Looking at the acceptance ratio plot (see Figure 13.67), however, suggests that the burn-in should be slightly larger. Therefore, we set a value of 50 for the burn-in using the `setBurnin` method (see *Investigating the quality of the samples through diagnostics and plots*). When a trace plot is drawn after defining a burn-in with `setBurnin`, the burn-in region is indicated by a black dotted line.

11.6.4.2 Drawing the acceptation ratio

This method is used to visualise the evolution of the acceptation ratio across iterations. It should be used to check the stability of the chains, since an unstable acceptation ratio indicates that the chains may not have converged. It also allows the user to verify that the acceptation ratio stabilizes within the desired range (between 20% and 50%), which ensures good exploration of the target density while retaining a sufficient number of samples.

Thus, acceptation ratio plots provide useful guidance for determining an appropriate **burn-in** (the number of iterations to discard before stability is reached) and for selecting a suitable initial standard deviation of the proposal distribution to achieve an acceptation ratio within the desired range. The prototype is:

```
drawAcceptationRatio(sTitre, variable = "*", option = "")
```

This method takes up to three arguments, two of which are optional:

1. **sTitre**: the title of the plot (an empty string is allowed);
2. **variable** (optional): a list of parameter names to be drawn, separated by colons ":". The default `"*"` draws all parameters;
3. **option** (optional): a list of options, separated by commas ",", to adjust the plotting behavior:
    - **"nonewcanvas"**: draw on the current canvas (instead of creating a new one);
    - **"vertical"**: if multiple parameters are plotted, display them stacked vertically (one per row). By default, plots are arranged horizontally, side by side;

Returning to the example in *Macro "calibrationMCMCFlowrate1D.py"*, the acceptation ratio plots (shown in Figure 13.67) reveal that the acceptation ratio is unstable during the first iterations, likely because the chains were initialised far from the most probable value. Afterward, the acceptation ratio stabilizes, oscillating around what appears to be an asymptote. Based on this, a burn-in of about 50 can be considered appropriate. When a acceptation ratio plot is drawn after defining a burn-in with `setBurnin`, the burn-in region is indicated by a black dotted line.

11.6.4.3 Checking for convergence with the Gelman–Rubin diagnostic

Once the trace and acceptance ratio plots have been analysed, and an initial burn-in value has been chosen, the Gelman–Rubin diagnostic can be used to verify the convergence of the chains. This diagnostic compares the intra- and inter-chain variances (see [Bla17] for more details).

The method `diagGelmanRubin` has the following prototype:

```
diagGelmanRubin()
```

This method takes no arguments, as it is called directly from the `TCalibration` object. For example, in *Macro "calibrationMCMCFlowrate1D.py"*, the Gelman–Rubin diagnostic is computed with:

```
GelmanRubin_values = cal.diagGelmanRubin()
GelmanRubin_values["hl"] # To extract the Gelman-Rubin statistic for parameter hl
```

The returned object `GelmanRubin_values` is an unordered map where each parameter name (e.g., `hl`) is associated with its Gelman–Rubin statistic value. The results are also printed in the console (see *Console*), accompanied by a short generic commentary interpreting the statistic.

In this example, the Gelman–Rubin diagnostic confirms good convergence of the chains (and thus the suitability of the chosen burn-in value). In other cases, where the statistics indicate poorer convergence, it may be necessary to run additional iterations (see *Running the estimate, exporting and loading chains, and continuing the calculation*) or to adjust the burn-in value (see *Investigating the quality of the samples through diagnostics and plots*).

11.6.4.4 Thinning the chains with ESS

Once convergence of the chains has been assessed using the trace plot (see *Drawing the trace*), the acceptance ratio plot (see *Drawing the acceptation ratio*), and the Gelman–Rubin diagnostic (see *Checking for convergence with the Gelman–Rubin diagnostic*), the next step is to evaluate the autocorrelation of the samples. If samples are highly correlated, the posterior distribution may not be properly explored (e.g., chains could remain trapped in a mode).

A common way to assess this is to compute the Effective Sample Size (ESS), which estimates how many samples can effectively be considered as independent among the available ones. ESS thus indirectly indicates the appropriate lag (the number of iterations to skip to obtain approximately uncorrelated samples). More details are provided in [Bla17].

The method `diagESS` has the following prototype:

```
diagESS()
```

This method takes no arguments, as it is called directly from the `TCalibration` object. For example, in *Macro "calibrationMCMCFlowrate1D.py"*, the ESS diagnostic is computed with:

```
ESS_values = cal.diagESS()
ESS_values["h1"][0] # To extract the ESS statistic for parameter h1 computed on the␣
↪first chain
```

The returned object `ESS_values` is an unordered map where each parameter name (e.g., `h1`) is associated with a vector of ESS values, one per chain. The results are also printed in the console (see *Console*).

In this example, the ESS diagnostic confirms that each parameter has several hundred (at least 200) effectively uncorrelated samples. It is therefore recommended to set the lag to 1 using `setLag` method (see *Investigating the quality of the samples through diagnostics and plots*), which indicates that the samples are sufficiently uncorrelated.

If the ESS is too small, it may be necessary to run additional iterations (see *Running the estimate, exporting and loading chains, and continuing the calculation*) or to adjust the initial standard deviation of the proposal distribution, which may be too small, causing the chain to move too slowly (see *Initialising the process*).

11.6.5 Looking at the results

Once the convergence of the chains has been assessed and thinning applied, the results can be analysed in more detail. While trace plots are already informative results on their own, several additional methods are available. Among the implemented tools, it is possible to display pairwise parameter traces (see *Drawing the 2D trace*), visualise the posterior distributions (see *Drawing the parameters*), and examine the residual distribution (see *Drawing the residuals*). These methods are described in the following sections:

11.6.5.1 Drawing the 2D trace

This method is used to visualise the trajectory of the chains across iterations by examining two parameters at a time. It can help verify that the chains uniformly explore the distribution and that they converge toward the same posterior. It also highlights potential covariance between pairs of parameters in the posterior distribution.

Although this method can be used as a diagnostic tool, it is less effective for assessing chain stability or detecting autocorrelation. For those purposes, standard trace plots remain preferable.

The prototype is:

```
draw2DTrace(sTitre, variable, option = "")
```

This method takes up to three arguments, two of which are optional:

1. **sTitre**: the title of the plot (an empty string is allowed);
2. **variable** (optional): a list of exactly two parameter names to be drawn, separated by colons `":"`. For example, `"t0:t1"` is a valid argument to plot the chains of `t0` and `t1` together on a 2D trace;
3. **option** (optional): a list of options, separated by commas ",", to adjust the plotting behavior:
    - **"nonewcanvas"**: draw on the current canvas (instead of creating a new one);

Returning to the example in *Macro "calibrationMCMCLinReg.py"*, the 2D trace plot (shown in Figure 13.72) clearly show the four chains converging toward the posterior distribution located in the center of the plot. The distribution appears to be uniformly covered by a sufficient number of samples, without any obvious trend suggesting covariance between the two parameters.

11.6.5.2 Drawing the parameters

The posterior distributions of the parameters can be visualised using the dedicated instance of `drawParameters`, whose prototype is identical to that discussed in *Drawing the parameters*.

```
void drawParameters(sTitre, variable = "*", option = "")
```

The only difference from the standard `drawParameters` method in `TCalibration` is that this version automatically accounts for the burn-in period and the lag (see *Investigating the quality of the samples through diagnostics and plots* for details). Specifically, the first samples of each chain are discarded according to the burn-in value, and the remaining samples are thinned using the specified lag (i.e., one sample is retained out of every lag iterations).

An example of this function is shown in Figure 13.74.

11.6.5.3 Drawing the residuals

The residuals can be visualised with the standard `drawResiduals` method of any `TCalibration` object. Its arguments and options have already been detailed in *Drawing the parameters*, and its prototype is recalled here for convenience:

```
drawResiduals(sTitre, variable = "*", select = "1>0", option = "")
```

Since no specific implementation has been added for MCMC calibration, the main point to clarify is how the *a priori* and *a posteriori* configurations are defined:

- ***A priori* configuration:** obtained from the parameter initialisation. This can either be provided explicitly by the user through the method described in *Initialising the process*, or, if not specified, generated from a random draw according to the given *a priori* probability density.
- ***A posteriori* configuration:** by default, the mean of the posterior distribution is used (as discussed in the constructor section of this class, see *Constructing the TMCMC object*). If the option **"mode"** is specified when constructing the `TMCMC` object, then the posterior mode is estimated and used instead.

An example of this function is shown in Figure 13.73.

Two examples are also provided in the use-case section (see *Macro "calibrationMCMCFlowrate1D.py"* and *Macro "calibrationMCMCLinReg.py"*).

11.7 CIRCE method

The CIRCE method is a statistical approach proposed as an alternative to expert judgement, designed to determine the uncertainty of parameters in a physical model. Such uncertainties are often difficult to assess because some parameters may not be directly measurable. However, by relying on separate-effect tests (SET) experiments, that are sensitive to the physical model, it becomes possible to infer estimates of these uncertainties.

As explained in *Introduction*, CIRCE is not a strict calibration method, its logic differs from the standard framework (see [Bla17] for details). For this reason, it does not inherit from the `TCalibration` class (and therefore does not provide the common calibration methods presented in *Common methods of the calibration classes*). Instead, it has its own dedicated structure, implemented in Uranie through the `TCirce` class.

The usage of the `TCirce` class can be summarised in a few key steps:

1. Prepare the data and the model:
   - Specify the experimental dataserver;
   - Construct the `TCirce` object by specifying the experimental attribute, the code attribute, and the sensitivity attributes (see *Constructing the TCirce object*).
2. Set the algorithm properties:
   - Initialise the algorithm;
   - Define optional behaviours (see *Defining the TCirce properties*).
3. Perform the estimate:
   - Run the estimate process (see *Running the estimate*).
4. Perform post-processing:
   - Extract the results for post-treatment (see *Looking at the results*).

11.7.1 Constructing the `TCirce` object

To create an instance of the `TCirce` class, the constructor is available with the following prototype:

```
TCirce(tds, systar, syhat, ssensi, sexpuncert = "", option = "")
```

This method takes up to six arguments, two of which are optional:

1. **tds**: a `TDataServer` object containing:
   - one attribute for the experimental data,
   - one attribute for the code output,
   - and the attributes corresponding to the derivatives of the code output with respect to each parameter (one attribute per parameter).
2. **systar**: name of the experimental attribute;
3. **syhat**: name of the code output attribute;
4. **ssensi**: list of attribute names corresponding to the derivatives of the code output with respect to each parameter, separated by commas ",";
5. **sexpuncert** (optional): name of the experimental uncertainty attribute;
6. **option** (optional): currently, no options are implemented for this class.

11.7.2 Defining the `TCirce` properties

Once the `TCirce` object has been constructed, it is possible to configure several hyperparameters of the algorithm.

The first one is the tolerance parameter used as a stopping criterion. Its **default value** is `1.0e-5`, but it can be modified with the `setTolerance` method, whose prototype is the following:

```
setTolerance(dtol)
```

This method accepts a single argument: a double precision value defining the new tolerance.

It is also possible to provide initial values for the algorithm:

- an initial vector of biases (**default value** is null vector), set with `setBVectorInitial`;
- an initial correlation matrix (**default value** is identity matrix), set with `setCMatrixInitial`.

```
setBVectorInitial(vec)
setCMatrixInitial(mat)
```

Each method accepts a single argument, namely the initial `TVectorD` (for the biases) or the initial `TMatrixD` (for the correlation matrix).

Finally, the user can define the number of iterations for running the algorithm with different initial correlation matrices (**default value** is 1) using the `setNCMatrix` method, whose prototype is the following:

```
setNCMatrix(n)
```

This method accepts a single integer argument that sets the number of iterations.

All these methods have corresponding getter functions, which allow retrieving the current values:

- `getTolerance` returns the tolerance (double),
- `getBVectorInitial` returns the initial bias vector (`TVectorD`),
- `getCMatrixInitial` returns the initial correlation matrix (`TMatrixD`),
- `getNCMatrix` returns the number of iterations (integer).

11.7.3 Running the estimate

Since the `TCirce` class does not inherit from the `TCalibration` class (as it is not strictly a calibration method), it provides its own execution method. This method is named `estimate` (rather than `estimateParameters`) and must be called directly from the `TCirce` object to compute the vector of biases and the correlation matrix. Its prototype is:

```
estimate(option = "")
```

This method accepts a single optional argument, which can specify options for running the process. At present, no options are implemented for this class, so the argument should not be used.

11.7.4 Looking at the results

Once the computation is complete, several methods are available to extract or display the results.

The bias vector and the correlation matrix can be retrieved with the `getBVector` and `getCMatrix` methods, respectively:

```
getBVector()
getCMatrix()
```

These methods take no arguments, as they are directly applied to the `TCirce` object after the process has been run. They return a `TVectorD` and a `TMatrixD` corresponding to the bias vector and the correlation matrix.

It is also possible to obtain the likelihood value for the optimal parameters with the `getLikelihood` method:

```
getLikelihood()
```

This method also takes no arguments. It returns a double corresponding to the likelihood value of the optimal parameters.

The methods `getMatrixVarianceMu` and `getMatrixVarianceSigma` return, respectively, the variance matrices of mu and sigma, as computed via the Fisher Matrix confidence interval:

```
getMatrixVarianceMu()
getMatrixVarianceSigma()
```

Finally, all of this information can be displayed on the console using the `printResults` method:

```
printResults(option = "")
```

This method accepts a single optional argument, which can specify options for running the process. At present, no options are implemented for this class, so the argument should not be used.

An example is also provided in the use-case section (see *Macro "calibrationCirce.py"*).

CHAPTER

TWELVE

THE UNCERTAINTY MODELER MODULE

Abstract This chapter presents the features of the UncertModeler module of Uranie - version v4.11.0. The *namespace* of this library is **URANIE::UncertModeler**.

12.1 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 [Bla17]. 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 12.1 shows the results of such a script.

```
"""
Example of distribution testing with different quality criteria
"""
from URANIE import DataServer, Sampler, UncertModeler
import ROOT

# Create a TDS with 3 kind of distributions
tds0 = DataServer.TDataServer()
tds0.addAttribute(DataServer.TNormalDistribution("n", 1.3, 4.5))
tds0.addAttribute(DataServer.TLogNormalDistribution("ln", 1.3, 4.5))
tds0.addAttribute(DataServer.TUniformDistribution("u", -1.3, 4.5))

# Create the sample
fsamp = Sampler.TBasicSampling(tds0, "lhs", 1000)
fsamp.generateSample()

# Create the canvas
c = ROOT.TCanvas("c1", "", 5, 20, 1300, 600)
apad = ROOT.TPad("apad", "apad", 0, 0.03, 1, 1)
apad.Draw()
apad.cd()
apad.Divide(3)

apad.cd(1)
tks_n = UncertModeler.TTestKolmogorovSmirnov(tds0, "n")
tcvm_n = UncertModeler.TTestCramerVonMises(tds0, "n")
tad_n = UncertModeler.TTestAndersonDarling(tds0, "n")
```

(continues on next page)

(continued from previous page)

```
# 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)
tks_ln = UncertModeler.TTestKolmogorovSmirnov(tds0, "ln")
tcvm_ln = UncertModeler.TTestCramerVonMises(tds0, "ln")
tad_ln = UncertModeler.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)
tks_u = UncertModeler.TTestKolmogorovSmirnov(tds0, "u")
tcvm_u = UncertModeler.TTestCramerVonMises(tds0, "u")
tad_u = UncertModeler.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
```

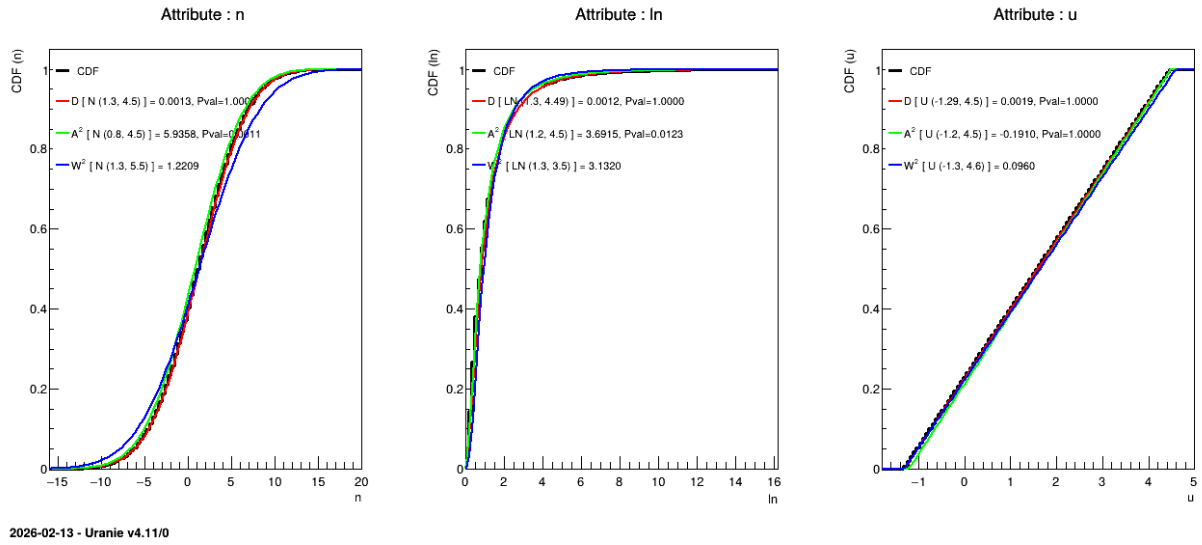


Figure 12.1: Results of the macro defined previously to produce variety of test of already implemented distributions

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.

CHAPTER

THIRTEEN

USE-CASES IN PYTHON

13.1 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.py` file, introduced in *Uranie namespace*, which is automatically loaded when executing python. An example would be:

```
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)
```

(continues on next page)

(continued from previous page)

```
# 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()
#
```

All the use case macros, shown in this chapter, are tested with `pylint` to check their consistency with python standard, such as PEP-8, most of which obtains now (from version 4.8.0) a score of 10 out of 10. To do this, three options are used:

- `--module-naming-style=camelCase`: this option is used so that evey file name can be written as their C++ counterpart (in order to overcome the default `sneak_case`);
- `--const-naming-style=any`: since our use-case macros are standalone scripts (meaning without classes) all their variables are global ones and should be written in upper case according to pylint default. We choose not to follow this convention;
- `--ignored-module=ROOT`: as `ROOT` is loaded from the `$PYTHONPATH` and not installed as a system library, many error might pop-up from the use of `ROOT`;

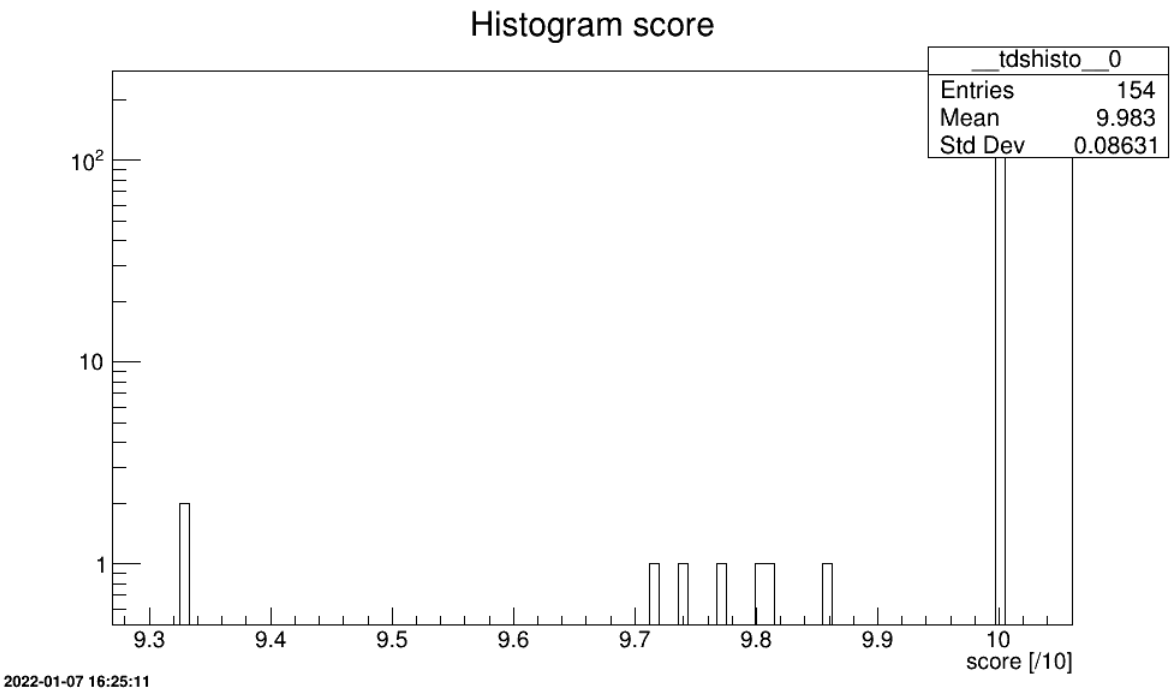


Figure 13.1: Distribution of the pylint score for the following use-case macros

13.2 Macros Python HowTo

In this section we propose to explain how to handle some of the ROOT or C++-based object in python. The idea is just to breakdown the already provided pieces of code shown in *FAQ: a Python handbook to interact with Uranie*.

13.2.1 Macro "howtoConvertTDataServerArray.py"

13.2.1.1 Objective

The goal of this macro is only to show how to convert a `TDataServer` into a `numpy.array` and vice-versa.

13.2.1.2 Macro Uranie

```
"""
Example of TDataServer into numpy array converter
"""
import numpy as np
from URANIE import DataServer

# import data into a dataserver
tds = DataServer.TDataServer("tds", "pouet")
tds.fileDataRead("myData.dat")
print("Dumping tds")
tds.scan()

# Define the list of variable to be read
VarList = "x:y"
```

(continues on next page)

(continued from previous page)

```
# Create an array by defining the shape and the type (this array is a matrix)
TdsData = np.empty(shape=(tds.getNPatterns(), len(VarList.split(":"))),
                   dtype=np.float64)
# Dump the data into the created array (the last field empty unless a select is done␣
↪on the sample)
tds.getTuple().extractData(TdsData, TdsData.size, VarList, tds.getCut().GetTitle())

# Check the content
print("Dumping the array")
print(TdsData)

# Create a new Dataserver and feed it
tds2 = DataServer.TDataServer("brandnew", "pouet")
# Add the attribute in the dataserver and create the tuple
for name in VarList.split(":"):
    index = VarList.split(":").index(name)
    VarData = np.ascontiguousarray(TdsData.transpose()[index])
    tds2.addAttributeUsingData(name, VarData, VarData.size)

print("Dumping new tds")
tds2.Scan("*")
```

The first block reads an input file `myData.dat` and store the data in the `TDataServer` as usually done. The new block is what we recommend: define the list of variable that one wants to store

```
VarList = "x:y"
```

Once done, the `np.array` has to be constructed by defining its content and size. Thanks to the `TDataServer` object and the list of variable this can be done easily as

```
TdsData = np.empty(shape=(tds.getNPatterns(), len(VarList.split(":"))),
                   dtype=np.float64)
```

Finally, the `extractData` method is called with the given created array and list of variables.

```
tds.getTuple().extractData(TdsData, TdsData.size, VarList, tds.getCut().GetTitle())
```

The second part consists in creating a brand new `TDataServer` from a `np.array`, which basically will consist in our case in re-creatting the original `TDataServer` with a different name and title. To do so, one starts by creating the `TDataServer`

```
tds2 = DataServer.TDataServer("brandnew", "pouet")
```

Then, one loops over the list of variable and in the loop two things are done :

- a new array is created by transposing the data, selecting the proper array of data and construct this as an memory-continuous array (for C++-consistency)
- a new attribute is created using the method `addAttributeUsingData` that needs the name of the attribute, the buffer and the size of the array provded

```
for name in VarList.split(":"):
    index = VarList.split(":").index(name)
```

(continues on next page)

(continued from previous page)

```
    VarData = np.ascontiguousarray(TdsData.transpose()[index])
    tds2.addAttributeUsingData(name, VarData, VarData.size)
```

The output of this macro is shown in *Console*.

13.2.1.3 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

Dumping tds
***********************************************
*    Row   * tds__n__i *       x.x *       y.y *
***********************************************
*        0 *         1 *        -5 *        25 *
*        1 *         2 *        -4 *        16 *
*        2 *         3 *        -3 *         9 *
*        3 *         4 *        -2 *         4 *
*        4 *         5 *        -1 *         1 *
*        5 *         6 *         0 *         0 *
*        6 *         7 *         1 *         1 *
*        7 *         8 *         2 *         4 *
*        8 *         9 *         3 *         9 *
*        9 *        10 *         4 *        16 *
*       10 *        11 *         5 *        25 *
***********************************************
Dumping the array
[[-5. 25.]
 [-4. 16.]
 [-3.  9.]
 [-2.  4.]
 [-1.  1.]
 [ 0.  0.]
 [ 1.  1.]
 [ 2.  4.]
 [ 3.  9.]
 [ 4. 16.]
 [ 5. 25.]]
Dumping new tds
***********************************************
*    Row   * brandnew_ *       x.x *       y.y *
***********************************************
*        0 *         1 *        -5 *        25 *
*        1 *         2 *        -4 *        16 *
*        2 *         3 *        -3 *         9 *
*        3 *         4 *        -2 *         4 *
*        4 *         5 *        -1 *         1 *
*        5 *         6 *         0 *         0 *
*        6 *         7 *         1 *         1 *
*        7 *         8 *         2 *         4 *
```

(continues on next page)

(continued from previous page)

```
*        8 *         9 *         3 *         9 *
*        9 *        10 *         4 *        16 *
*       10 *        11 *         5 *        25 *
*************************************************
```

13.2.2 Macro "howtoConvertTMatrixDArray.py"

13.2.2.1 Objective

The goal of this macro is only to show how to convert a `TMatrixD` object into a `numpy.array` and vice-versa.

13.2.2.2 Macro Uranie

```
"""
Example of TMatrix into numpy array conversion
"""
import numpy as np
import ROOT

# define the number of rows and columns
nrow = 3
ncol = 5

# Initialise and fill a TMatrixD
InMat = ROOT.TMatrixD(nrow, ncol)
for i in range(nrow):
    for j in range(ncol):
        InMat[i][j] = ROOT.gRandom.Gaus(0, 1)

print("Original TMatrixD")
InMat.Print()

# Create the ndarray with the good shape
mat_version = np.frombuffer(InMat.GetMatrixArray(), dtype=np.float64,
                            count=nrow*ncol).reshape(nrow, ncol)
print("Numpy array transformation")
print(mat_version)

# Back to another TMatrixD
OutMat = ROOT.TMatrixD(nrow, ncol)
OutMat.SetMatrixArray(mat_version)
print("\nRecreated TMatrixD")
OutMat.Print()
```

The first block simply defines the size of the matrix to be created and creates along, from scratch, a `TMatrixD` with a random gaussian content. Once done, the conversion into a `np.array` is done in a single line manner below that would be breakdown to get the full picture:

```
mat_version = np.frombuffer(InMat.GetMatrixArray(), dtype=np.float64,
                            count=nrow*ncol).reshape(nrow, ncol)
```

The content of the original `TMatrixD` can be accessed by calling `GetMatrixArray()`. This returns a pointer to the first element's address, so it can be used by the lowest level array object of numpy, the `np.frombuffer` which also needs the

type of data and the number of elements. This is important as the memory will be scanned given these two information. The last step is to call, on this newly created object the `reshape` method to create a properly organised `np.array`.

The second part consists in creating a brand new `TMatrixD` from the newly created `np.array`. The idea is simply to create the object empty and use the `SetMatrixArray` method to dump the array content into the matrix, as done below.

```
OutMat = ROOT.TMatrixD(nrow, ncol)
OutMat.SetMatrixArray(mat_version)
```

The output of this macro is shown in *Console*.

13.2.2.3 Console

```
Original TMatrixD

3x5 matrix is as follows

     |      0    |      1    |      2    |      3    |      4    |
-------------------------------------------------------------------
   0 |     0.9989     -0.4348      0.7818    -0.03005      0.8243
   1 |   -0.05672     -0.9009     -0.0747    0.007912     -0.4108
   2 |      1.391     -0.9851    -0.04894      -1.443      -1.061

Numpy array transformation
[[ 0.99893272 -0.43476439  0.78179626 -0.03005277  0.82426369]
 [-0.05671733 -0.90087599 -0.07470447  0.00791221 -0.41076317]
 [ 1.39119397 -0.98506611 -0.04894054 -1.44333537 -1.06067041]]

Recreated TMatrixD

3x5 matrix is as follows

     |      0    |      1    |      2    |      3    |      4    |
-------------------------------------------------------------------
   0 |     0.9989     -0.4348      0.7818    -0.03005      0.8243
   1 |   -0.05672     -0.9009     -0.0747    0.007912     -0.4108
   2 |      1.391     -0.9851    -0.04894      -1.443      -1.061
```

13.2.3 Macro “howtoLoadFunction.py”

13.2.3.1 Objective

The goal of this macro is only to show how to load a C++ function in python as this could be of use for instance when producing a surrogate model (all of them can be exported in C++ and even though the recommended way to handle these is through the Uranie classes, it might happen that one wants to put one’s hand on it).

Our example is taken from the `UserFunction.C` file provided with the Uranie sources (look for it in the the `"$URANIESYS/share/uranie/macros"` folder). The `operation` function only compute the product, ratio, sum and difference between two provided inputs (just checking for consistency that the denominator of the ratio is not equal to zero). The code is shown here for illustration purpose.

```
void operation(double *x, Double_t *y)
{
```

(continues on next page)

(continued from previous page)

```
  Double_t x1 = x[0], x2 = x[1];
  // product
  y[0] = x1 * x2;
  // divide
  y[1] = ((x2!=0) ? x1 / x2 : -999);
  // sum
  y[2] = x1 + x2;
  // diff
  y[3] = x1 - x2;
}
```

13.2.3.2 Macro Uranie

```
"""
Example of C++ function in python
"""
import numpy as np
import ROOT

# Loading a function
ROOT.gROOT.LoadMacro("UserFunctions.C")

# Preparing inputs and outputs
inp = np.array([1.2, 0.8])  # input values
out = np.zeros(4)  # output values initialise with 4 zeros

# Call the method, through the ROOT interface
ROOT.operation(inp, out)

# Print the result
print("Results are")
print(out)
```

The first block simply loads the input file `UserFunctions.C` through the ROOT's interface.

```
ROOT.gROOT.LoadMacro("UserFunctions.C")
```

Once done, the input and output `np.array` object are created, the former with provided values to be tested, the latter only filled with zeros. Calling the function is simply done by providing input and output arrays to the function operator that is now stored within ROOT by doing:

```
ROOT.operation(inp, out)
```

The output of this macro is shown in *Console*.

13.2.3.3 Console

```
Results are
[0.96 1.5  2.   0.4 ]
```

13.2.4 Macro "howtoPassReference.py"

13.2.4.1 Objective

The goal of this macro is to show how to use function that would have been defined using the "by-reference" prototype, which is specific to C++. A simple exemple in Uranie is provided when considering the `computeQuantile` method: the second argument is a double in which the result will be stored.

13.2.4.2 Macro Uranie

```
"""
Example of C++style reference usage
"""
from ctypes import c_double  # for ROOT version greater or equal to 6.20
from URANIE import DataServer

# create a dataserver and read data
tds = DataServer.TDataServer("pouet", "foo")
tds.fileDataRead("myData.dat")

# compute quantile (this method get the results 'by reference')
proba = 0.9  # ROOT.Double(0.9) for ROOT version lower than 6.20
quant = c_double(0.0)  # ROOT.Double(0.0) for ROOT version lower than 6.20

# Compute the quantile and dump the value returned by reference
tds.computeQuantile("x", proba, quant)
print("Result is ")
print(quant.value)
```

It starts by calling the `ctypes` module in order to get the `c_double` interface. This is only for ROOT version greater than 6.20. For older version, the solution relied on ROOT only (calling another interface discussed later-on).

```
from ctypes import c_double  # for ROOT version greater or equal to 6.20
```

Once done, a `TDataServer` object is created and filled from the `myData.dat` file. One then wants to estimate the 90% quantile, which can be done by calling the `computeQuantile` method, which requires two double as input, both passed "by-reference". To do so, one has to create two specific objects as below:

```
proba = 0.9  # ROOT.Double(0.9) for ROOT version lower than 6.20
quant = c_double(0.0)  # ROOT.Double(0.0) for ROOT version lower than 6.20
```

Once done, the method is called and the result is displayed by getting the value of the `c_double` object

```
print(quant.value)
```

The output of this macro is shown in *Console*.

13.2.4.3 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026
```

(continues on next page)

(continued from previous page)

```
Result is
4.0
```

13.3 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.

13.3.1 Macro "dataserverAttributes.py"

13.3.1.1 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`.

13.3.1.2 Macro Uranie

```
"""
Example of attribute management
"""
from URANIE import DataServer

# Define the attribute "x"
px = DataServer.TAttribute("x", -2.0, 4.0)
px.setTitle("#Delta P^{#sigma}")
px.setUnity("#frac{mm^{2}}{s}")

# Define the attribute "y"
py = DataServer.TAttribute("y", 0.0, 1.0)

# Define the DataServer of the study
tds = DataServer.TDataServer("tds", "my first TDS")

# Add the attributes in the DataServer.TDataServer
tds.addAttribute(px)
tds.addAttribute(py)
tds.addAttribute(DataServer.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 $\frac{mm^2}{s}$

```
px = DataServer.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.

```
py = DataServer.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.

```
tds = DataServer.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(DataServer.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.

13.3.1.3 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

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[]
```

(continues on next page)

(continued from previous page)

```
** 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]
```

(continues on next page)

(continued from previous page)

```
**   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[]
```

13.3.2 Macro “dataserverMerge.py”

13.3.2.1 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
```

(continues on next page)

(continued from previous page)

```
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.

13.3.2.2 Macro Uranie

```
"""
Example of data merging
"""
from URANIE import DataServer

tds1 = DataServer.TDataServer()
tds2 = DataServer.TDataServer()

tds1.fileDataRead("tds1.dat")
print("Dumping tds1")
tds1.Scan("*")

tds2.fileDataRead("tds2.dat")
print("Dumping tds2")
tds2.Scan("*")

tds1.merge(tds2)
print("Dumping merged tds1 and tds2")
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("*")
```

13.3.2.3 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

Dumping tds1
************************************************************************
*    Row   * tds1__n__ *       x.x *     dy.dy *       z.z * theta.the *
```

(continues on next page)

(continued from previous page)

```
*************************************************************************
*        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 *
*************************************************************************
```

13.3.3 Macro “dataserverLoadASCIIFilePasture.py”

13.3.3.1 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
```

(continues on next page)

(continued from previous page)

```
21 18.59
28 22.33
42 39.35
57 56.11
63 61.73
70 64.62
79 67.08
```

13.3.3.2 Macro Uranie

```
"""
Example of data loading with pasture file
"""
from URANIE import DataServer, Sampler, Launcher
import ROOT

C = ROOT.TCanvas("mycanvas", "mycanvas", 1)
ROOT.gROOT.LoadMacro("UserFunctions.C")

tds = DataServer.TDataServer()
tds.fileDataRead("pasture.dat")

tds.getTuple().SetMarkerStyle(8)
tds.getTuple().SetMarkerSize(1.5)
tds.draw("yield:time")

tlf = Launcher.TLauncherFunction(tds, "ModelPasture", "time", "yhat")
tlf.run()

tds.getTuple().SetMarkerColor(ROOT.kBlue)
tds.getTuple().SetLineColor(ROOT.kBlue)

tds.draw("yhat:time", "", "lpsame")

tds2 = DataServer.TDataServer()
tds2.addAttribute(DataServer.TUniformDistribution("time2", 9, 80))

tsamp = Sampler.TSampling(tds2, "lhs", 1000)
tsamp.generateSample()

tds2.getTuple().SetMarkerColor(ROOT.kGreen)
tds2.getTuple().SetLineColor(ROOT.kGreen)
tlf = Launcher.TLauncherFunction(tds2, "ModelPasture", "", "yhat2")
tlf.run()

tds2.draw("yhat2:time2", "", "psame")
tds.draw("yhat:time", "", "lpsame")

ROOT.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])));
}
```

Which is C++ and is loaded thanks to the function `ROOT.gROOT.LoadMacro("UserFunctions.C")`.

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*.

```
tlf = Launcher.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.

```
tsamp = Sampler.TSampling(tds2, "lhs", 1000)
tsamp.generateSample()
```

The design is now evaluated with this `TDataServer` on the attribute **time2**

```
tlf = Launcher.TLauncherFunction(tds2, "ModelPasture", "", "yhat2")
tlf.run()
```

13.3.3.3 Graph

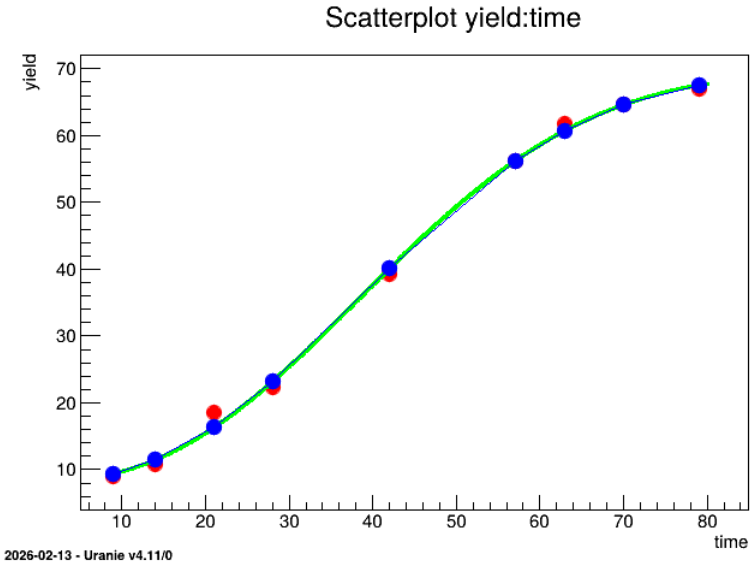


Figure 13.2: Graph of the macro `"dataserverLoadASCIIFilePasture.py"`

13.3.3.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

Info in <TCanvas::Print>: png file pasture.png has been created
```

13.3.4 Macro "dataserverLoadASCIIFile.py"

13.3.4.1 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
```

(continues on next page)

(continued from previous page)

```
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
```

13.3.4.2 Macro Uranie

```
"""
Example of data loading file
"""
from URANIE import DataServer
import ROOT

# Create a DataServer.TDataServer
tds = DataServer.TDataServer()
# Load the data base in the DataServer
tds.fileDataRead("flowrateUniformDesign.dat")

# Graph
Canvas = ROOT.TCanvas("c1", "Graph for the Macro loadASCIIFile",
                      5, 64, 1270, 667)
pad = ROOT.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()
```

13.3.4.3 Graph

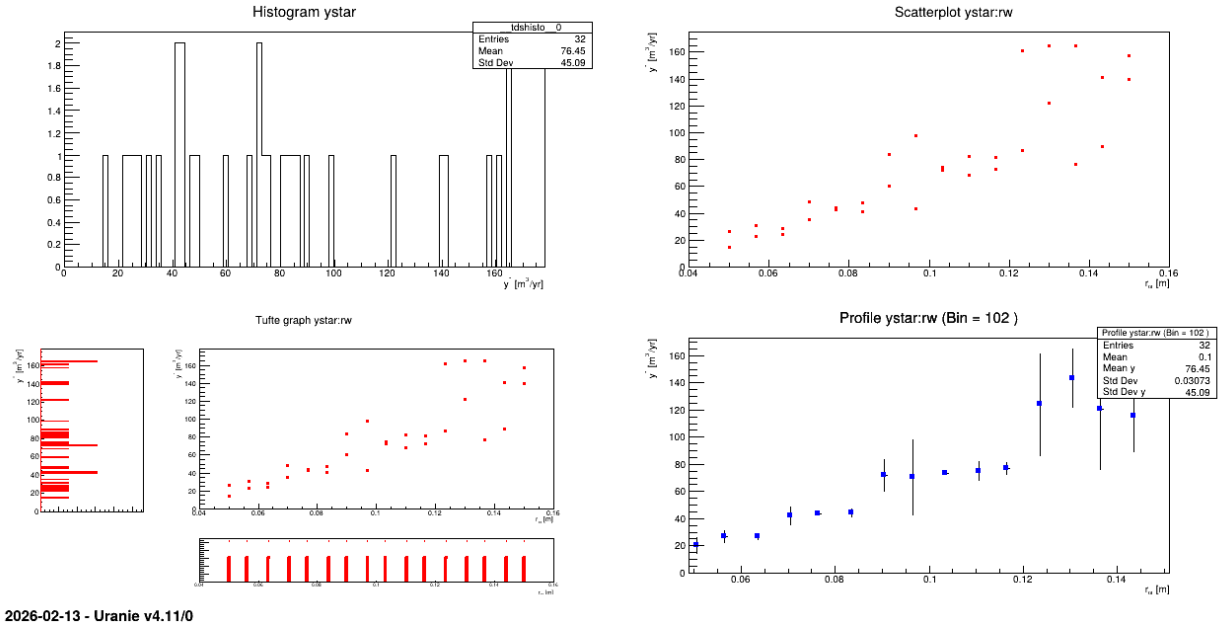


Figure 13.3: Graph of the macro "dataserverLoadASCIIFile.py"

13.3.5 Macro “dataserverLoadASCIIFileYoungsModulus.py”

13.3.5.1 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
```

(continues on next page)

(continued from previous page)

```
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,
```

(continues on next page)

(continued from previous page)

```
 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
```

13.3.5.2 Macro Uranie

```
"""
Example of young modulus data loading
"""
from URANIE import DataServer
import ROOT

tds = DataServer.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")
```

(continues on next page)

(continued from previous page)

```
Canvas = ROOT.TCanvas("c1", "Graph for the Macro loadASCIIFile",
                      5, 64, 1270, 667)
pad = ROOT.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")
```

13.3.5.3 Graph

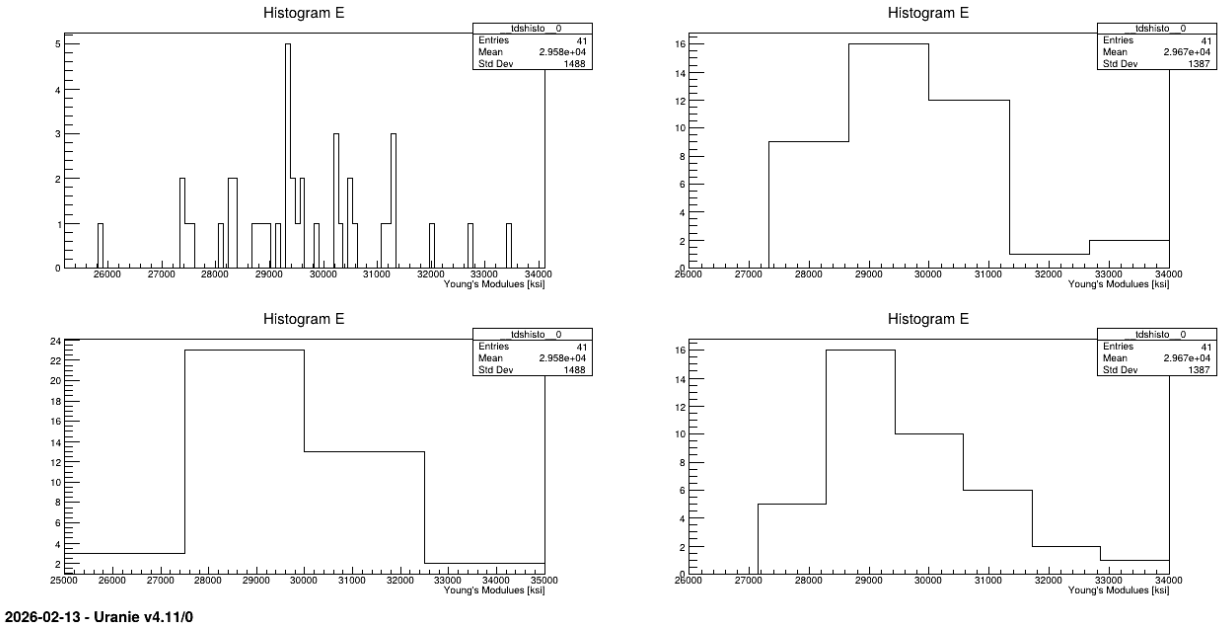


Figure 13.4: Graph of the macro "`dataserverLoadASCIIFileYoungsModulus.py`"

13.3.5.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

*******************************
** 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]
```

(continues on next page)

(continued from previous page)

```
*******************************
```

13.3.6 Macro "dataserverLoadASCIIFileIonosphere.py"

13.3.6.1 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.

13.3.6.2 Macro Uranie

```
"""
Example of data loading for Ionosphere dataset
"""
from URANIE import DataServer
import ROOT

tds = DataServer.TDataServer()
tds.fileDataRead("ionosphere.dat")

tds.getAttribute("x28").SetTitle("#Delta P_{e}^{F_{iso}}")

# Graph
Canvas = ROOT.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")
```

13.3.6.3 Graph

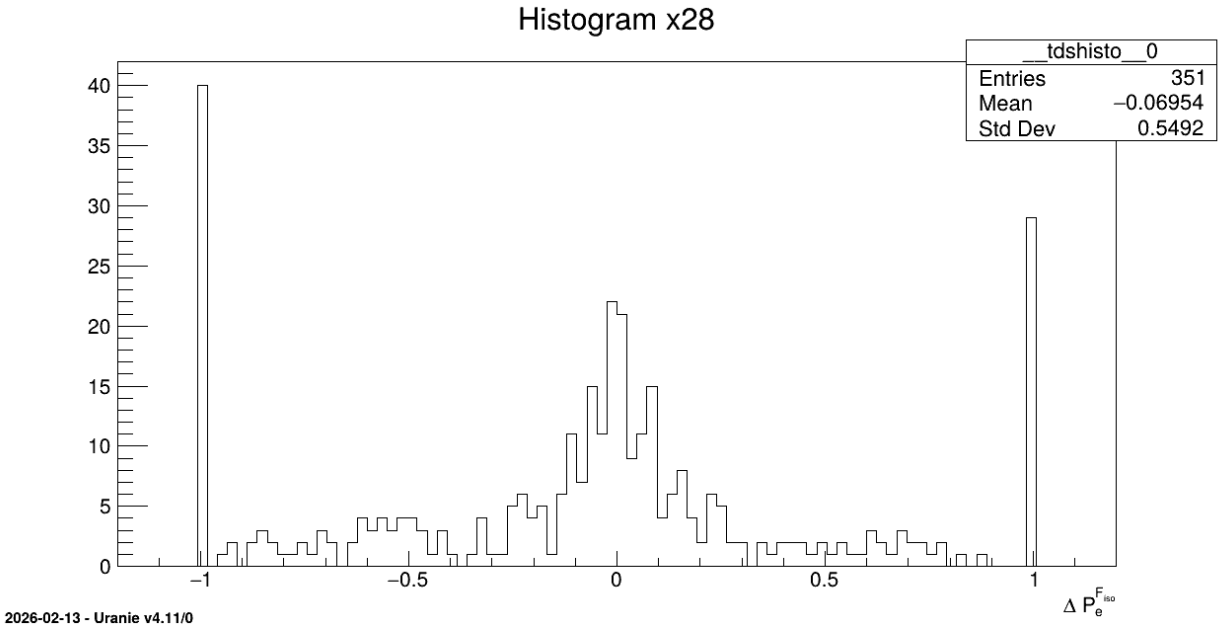


Figure 13.5: Graph of the macro "dataserverLoadASCIIFileIonosphere.py"

13.3.7 Macro "dataserverLoadASCIIFileCornell.py"

13.3.7.1 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
```

13.3.7.2 Macro Uranie

```
"""
Example of data loading and stat analysis
"""
from URANIE import DataServer

tds = DataServer.TDataServer()
tds.fileDataRead("cornell.dat")

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:

```
matCorr = tds.computeCorrelationMatrix("")
```

13.3.7.3 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026


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
```

13.3.8 Macro “dataserverComputeQuantile.py”

13.3.8.1 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).

13.3.8.2 Macro Uranie

```
"""
Example of quantile estimation (Wilks and not) for illustration purpose
"""
from ctypes import c_double  # For ROOT version greater or equal to 6.20
import numpy as np
from URANIE import DataServer
import ROOT

# Create a DataServer
tds = DataServer.TDataServer("foo", "pouet")
tds.addAttribute("x")  # With one attribute

# Create Histogram to store the quantile values
Q200 = ROOT.TH1F("quantile200", "", 60, 1, 4)
Q200.SetLineColor(1)
Q200.SetLineWidth(2)
Q400 = ROOT.TH1F("quantile400", "", 60, 1, 4)
Q400.SetLineColor(4)
Q400.SetLineWidth(2)
QW95 = ROOT.TH1F("quantileWilks95", "", 60, 1, 4)
QW95.SetLineColor(2)
QW95.SetLineWidth(2)
QW95400 = ROOT.TH1F("quantileWilks95400", "", 60, 1, 4)
QW95400.SetLineColor(8)
QW95400.SetLineWidth(2)
QW99 = ROOT.TH1F("quantileWilks99", "", 60, 1, 4)
QW99.SetLineColor(6)
QW99.SetLineWidth(2)

# Defining the sample size
nb = np.array([200, 400, 59, 90])
proba = 0.95  # Quantile value
CL = np.array([0.95, 0.99])  # Confidence level value for the two Wilks computation
# Loop over the number of estimation
for iq in range(4):
```

(continues on next page)

(continued from previous page)

```
    # Produce 10000 drawing to get smooth distribution
    for itest in range(10000):

        tds.createTuple()  # Create the tuple to store value
        # Fill it with random drawing of centered gaussian
        for ival in range(nb[iq]):
            tds.getTuple().Fill(ival+1, ROOT.gRandom.Gaus(0, 1))

        # Estimate the quantile...
        quant = c_double(0)  # ROOT.Double for ROOT version lower than 6.20
        if iq < 2:

            # ... with usual methods...
            tds.computeQuantile("x", proba, quant)
            if iq == 0:
                Q200.Fill(quant.value)  # ... on a 200-points sample
            else:
                Q400.Fill(quant.value)  # ... on a 400-points sample
                tds.estimateQuantile("x", proba, quant, CL[iq-1])
                QW95400.Fill(quant.value)  # compute the quantile at 95% CL

        else:

            # ... with the wilks optimised sample
            tds.estimateQuantile("x", proba, quant, CL[iq-2])
            if iq == 2:
                QW95.Fill(quant.value)  # compute the quantile at 95% CL
            else:
                QW99.Fill(quant.value)  # compute the quantile at 99% CL

        # Delete the tuple
        tds.deleteTuple()

# Produce the plot with requested style
ROOT.gStyle.SetOptStat(0)
can = ROOT.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
lin = ROOT.TLine()
lin.SetLineStyle(3)
lin.DrawLine(1.645, 0, 1.645, Q400.GetMaximum())

# Add a block of legend
leg = ROOT.TLegend(0.4, 0.6, 0.8, 0.85)
leg.AddEntry(lin, "Theoretical quantile", "l")
```

(continues on next page)

(continued from previous page)

```
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
nb = np.array([200, 400, 59, 90])
proba = 0.95  # Quantile value
CL = np.array([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.

13.3.8.3 Graph

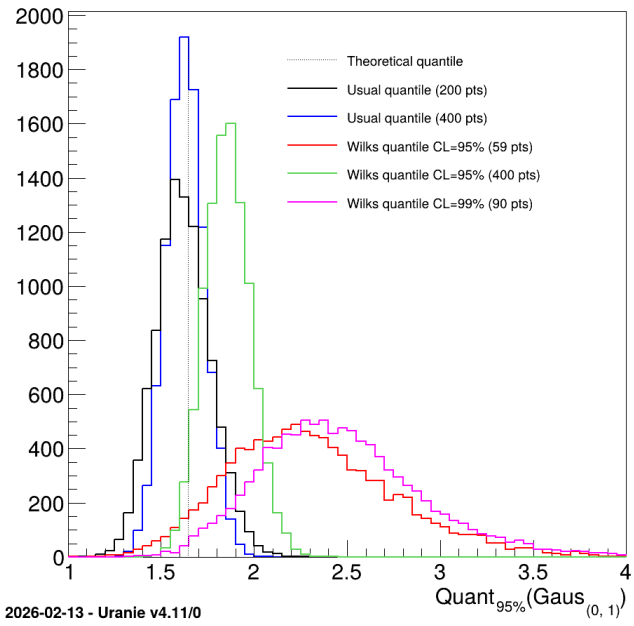


Figure 13.6: Graph of the macro "`dataserverComputeQuantile.py`"

13.3.9 Macro “dataserverGeyserStat.py”

13.3.9.1 Objective

This part shows the complete code used to produce the console display in *Computing the elementary statistic*.

13.3.9.2 Macro Uranie

```
"""
Example of statistical usage
"""
from URANIE import DataServer

tdsGeyser = DataServer.TDataServer("geyser", "poet")
tdsGeyser.fileDataRead("geyser.dat")
tdsGeyser.computeStatistic("x1")

print("min(x1)= "+str(tdsGeyser.getAttribute("x1").getMinimum())+";  max(x1)= "
      + str(tdsGeyser.getAttribute("x1").getMaximum())+";  mean(x1)= " +
      str(tdsGeyser.getAttribute("x1").getMean())+";  std(x1)= " +
      str(tdsGeyser.getAttribute("x1").getStd()))
```

13.3.9.3 Console

This macro should result in this output in console:

```
min(x1)= 1.6;  max(x1)= 5.1;  mean(x1)= 3.487783088235295;  std(x1)= 1.141371251105209
```

13.3.10 Macro “dataserverGeyserRank.py”

13.3.10.1 Objective

This part shows the complete code used to produce the console display in *Computing the ranking*.

13.3.10.2 Macro Uranie

```
"""
Example of rank usage for illustration purpose
"""
from URANIE import DataServer

tdsGeyser = DataServer.TDataServer("geyser", "poet")
tdsGeyser.fileDataRead("geyser.dat")
tdsGeyser.computeRank("x1")
tdsGeyser.computeStatistic("Rk_x1")

print("NPatterns="+str(tdsGeyser.getNPatterns())+";  min(Rk_x1)= " +
      str(tdsGeyser.getAttribute("Rk_x1").getMinimum())+";  max(Rk_x1)= " +
      str(tdsGeyser.getAttribute("Rk_x1").getMaximum()))
```

13.3.10.3 Console

This macro should result in this output in console:

```
NPatterns=272;  min(Rk_x1)= 1.0;  max(Rk_x1)= 272.0
```

13.3.11 Macro “dataserverNormaliseVector.py”

13.3.11.1 Objective

This part shows the complete code used to produce the console display in *Normalising the variable*.

13.3.11.2 Macro Uranie

```
"""
Example of vector normalisation
"""
from URANIE import DataServer

tdsop = DataServer.TDataServer("foo", "pouet")
tdsop.fileDataRead("tdstest.dat")

# Compute a global normalisation of v, CenterReduced
tdsop.normalize("v", "GCR", DataServer.TDataServer.kCR, True)
# Compute a normalisation of v, CenterReduced (not global but entry by entry)
tdsop.normalize("v", "CR", DataServer.TDataServer.kCR, False)

# Compute a global normalisation of v, Centered
tdsop.normalize("v", "GCent", DataServer.TDataServer.kCentered)
# Compute a normalisation of v, Centered  (not global but entry by entry)
tdsop.normalize("v", "Cent", DataServer.TDataServer.kCentered, False)

# Compute a global normalisation of v, ZeroOne
tdsop.normalize("v", "GZO", DataServer.TDataServer.kZeroOne)
# Compute a normalisation of v, ZeroOne (not global but entry by entry)
tdsop.normalize("v", "ZO", DataServer.TDataServer.kZeroOne, False)

# Compute a global normalisation of v, MinusOneOne
tdsop.normalize("v", "GMOO", DataServer.TDataServer.kMinusOneOne, True)
# Compute a normalisation of v, MinusOneOne (not global but entry by entry)
tdsop.normalize("v", "MOO", DataServer.TDataServer.kMinusOneOne, False)

tdsop.scan("v:vGCR:vCR:vGCent:vCent:vGZO:vZO:vGMOO:vMOO", "",
           "colsize=4 col=2:5::::::::")
```

13.3.11.3 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 *
```

(continues on next page)

(continued from previous page)

```
*        2 *        2 *  9 * 1.460 *    1 *    4 *    3 *    1 *    1 *    1 *    1 *
*******************************************************************************
```

13.3.12 Macro “dataserverComputeStatVector.py”

13.3.12.1 Objective

This part shows the complete code used to produce the console display in *Specific case of vectors*.

13.3.12.2 Macro Uranie

```
"""
Example of statistical estimation on vector attributes
"""
from URANIE import DataServer

tdsop = DataServer.TDataServer("foo", "poet")
tdsop.fileDataRead("tdstest.dat")

# Considering every element of a vector independent from the others
tdsop.computeStatistic("x")
px = tdsop.getAttribute("x")

print("min(x[0])= "+str(px.getMinimum(0))+";  max(x[0])= "+str(px.getMaximum(0))
      + ";  mean(x[0])= "+str(px.getMean(0))+";  std(x[0])= "+str(px.getStd(0)))
print("min(x[1])= "+str(px.getMinimum(1))+";  max(x[1])= "+str(px.getMaximum(1))
      + ";  mean(x[1])= "+str(px.getMean(1))+";  std(x[1])= "+str(px.getStd(1)))
print("min(x[2])= "+str(px.getMinimum(2))+";  max(x[2])= "+str(px.getMaximum(2))
      + ";  mean(x[2])= "+str(px.getMean(2))+";  std(x[2])= "+str(px.getStd(2)))
print("min(xtot)= "+str(px.getMinimum(3))+";  max(xtot)= "+str(px.getMaximum(3))
      + ";  mean(xtot)= "+str(px.getMean(3))+";  std(xtot)= "+str(px.getStd(3)))

# 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:::6")
```

13.3.12.3 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])= 1.0;  max(x[0])= 7.0;  mean(x[0])= 3.0;  std(x[0])= 3.4641016151377544
min(x[1])= 2.0;  max(x[1])= 8.0;  mean(x[1])= 4.666666666666666;  std(x[1])= 3.
↪055050463303893
min(x[2])= 3.0;  max(x[2])= 9.0;  mean(x[2])= 6.666666666666666;  std(x[2])= 3.
↪2145502536643185
min(xtot)= 1.0;  max(xtot)= 9.0;  mean(xtot)= 4.777777777777778;  std(xtot)= 3.
↪2317865716108862
```

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 *
*******************************************************
```

13.3.13 Macro “dataserverComputeCorrelationMatrixVector.py”

13.3.13.1 Objective

This part shows the complete code used to produce the console display in *Special case of vector*.

13.3.13.2 Macro Uranie

```
"""
Example of correlation matrix computation for vector
"""
from URANIE import DataServer

tdsop = DataServer.TDataServer("foo", "poet")
tdsop.fileDataRead("tdstest.dat")

# Consider a and x attributes (every element of the vector)
globalOne = tdsop.computeCorrelationMatrix("x:a")
globalOne.Print()

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

13.3.13.3 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
```

(continues on next page)

(continued from previous page)

```
2x2 matrix is as follows

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

13.3.14 Macro "dataserverComputeQuantileVec.py"

13.3.14.1 Objective

This part shows the complete code used to produce the console display in *computeQuantile*.

13.3.14.2 Macro Uranie

```
"""
Example of quantile estimation for many values at once
"""
from sys import stdout
from ctypes import c_int, c_double
import numpy as np
from URANIE import DataServer
import ROOT

tdsvec = DataServer.TDataServer("foo", "bar")
tdsvec.fileDataRead("aTDSWithVectors.dat")

probas = np.array([0.2, 0.6, 0.8], 'd')
quants = np.array(len(probas)*[0.0], 'd')
tdsvec.computeQuantile("rank", len(probas), probas, quants)

prank = tdsvec.getAttribute("rank")
nbquant = c_int(0)
prank.getQuantilesSize(nbquant)  # (1)
print("nbquant = " + str(nbquant.value))

aproba = c_double(0.8)
aquant = c_double(0)
prank.getQuantile(aproba, aquant)  # (2)
print("aproba = " + str(aproba.value) + ", aquant = " + str(aquant.value))

theproba = np.array(nbquant.value*[0.0], 'd')
thequant = np.array(nbquant.value*[0.0], 'd')
prank.getQuantiles(theproba, thequant)  # (3)
for i_q in range(nbquant.value):
    print("(theproba, thequant)[" + str(i_q) + "] = (" + str(theproba[i_q]) +
          ", " + str(thequant[i_q]) + ")")

allquant = ROOT.vector('double')()
prank.getQuantileVector(aproba, allquant)  # (4)
stdout.write("aproba = " + str(aproba.value) + ", allquant = ")
```

(continues on next page)

(continued from previous page)

```
for quant_i in allquant:
    stdout.write(str(quant_i) + " ")
print("")
```

13.3.14.3 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
```

13.3.15 Macro “dataserverDrawQQPlot.py”

13.3.15.1 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.

13.3.15.2 Macro Uranie

```
"""
Example of QQ plot
"""
from URANIE import DataServer, Sampler
import ROOT

# Create a TDS with 8 kind of distributions
p1 = 1.3
p2 = 4.5
p3 = 0.9
p4 = 4.4  # Fixed values for parameters

tds0 = DataServer.TDataServer()
tds0.addAttribute(DataServer.TNormalDistribution("norm", p1, p2))
tds0.addAttribute(DataServer.TLogNormalDistribution("logn", p1, p2))
tds0.addAttribute(DataServer.TUniformDistribution("unif", p1, p2))
tds0.addAttribute(DataServer.TExponentialDistribution("expo", p1, p2))
tds0.addAttribute(DataServer.TGammaDistribution("gamm", p1, p2, p3))
tds0.addAttribute(DataServer.TBetaDistribution("beta", p1, p2, p3, p4))
tds0.addAttribute(DataServer.TWeibullDistribution("weib", p1, p2, p3))
tds0.addAttribute(DataServer.TGumbelMaxDistribution("gumb", p1, p2))

# Create the sample
fsamp = Sampler.TBasicSampling(tds0, "lhs", 200)
fsamp.generateSample()

# Define number of laws, their name and numbers of parameters
nLaws = 8
```

(continues on next page)

(continued from previous page)

```
# number of parameters to put in () for the corresponding law
laws = ["normal", "lognormal", "uniform", "gamma", "weibull",
        "beta", "exponential", "gumbelmax"]
npar = [2, 2, 2, 3, 3, 4, 2, 2]

# Create the canvas
c = ROOT.TCanvas("c1", "", 800, 1000)
# Create the 8 pads
apad = ROOT.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
nS = 1000
mod = 0.8  # Factor used to artificially change the parameter values

Par = lambda i, n, p, mod: "," + str(p*mod) if n >= i else ""
opt = ""  # option of the drawQQPlot method
for i in range(nLaws):

    # Clean sstr
    test = ""
    # Add nominal configuration
    test += laws[i] + "(" + str(p1) + "," + str(p2) + Par(3, npar[i], p3, 1) \
        + str(p2) + Par(4, npar[i], p4, 1) + ")"
    # Changing par1
    test += ":" + laws[i] + "(" + str(p1*mod) + "," + str(p2) \
        + Par(3, npar[i], p3, 1) + str(p2) + Par(4, npar[i], p4, 1) + ")"
    # Changing par2
    test += ":" + laws[i] + "(" + str(p1) + "," + str(p2*mod) \
        + Par(3, npar[i], p3, 1) + str(p2) + Par(4, npar[i], p4, 1) + ")"
    # Changing par3
    if npar[i] >= 3:
        test += ":" + laws[i] + "(" + str(p1) + "," + str(p2) \
            + Par(3, npar[i], p3, mod) + str(p2) + Par(4, npar[i], p4, 1) + ")"
    # Changing par4
    if npar[i] >= 4:
        test += ":" + laws[i] + "(" + str(p1) + "," + str(p2) \
            + Par(3, npar[i], p3, 1) + str(p2) + Par(4, npar[i], p4, mod) + ")"

    apad.cd(i+1)
    # Produce the plot
    tds0.drawQQPlot(laws[i][:4], test, 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
p1 = 1.3
p2 = 4.5
```

(continues on next page)

(continued from previous page)

```
p3 = 0.9
p4 = 4.4  # Fixed values for parameters

tds0 = DataServer.TDataServer()
tds0.addAttribute(DataServer.TNormalDistribution("norm", p1, p2))
tds0.addAttribute(DataServer.TLogNormalDistribution("logn", p1, p2))
tds0.addAttribute(DataServer.TUniformDistribution("unif", p1, p2))
tds0.addAttribute(DataServer.TExponentialDistribution("expo", p1, p2))
tds0.addAttribute(DataServer.TGammaDistribution("gamm", p1, p2, p3))
tds0.addAttribute(DataServer.TBetaDistribution("beta", p1, p2, p3, p4))
tds0.addAttribute(DataServer.TWeibullDistribution("weib", p1, p2, p3))
tds0.addAttribute(DataServer.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
nLaws = 8
# number of parameters to put in () for the corresponding law
laws = ["normal", "lognormal", "uniform", "gamma", "weibull",
        "beta", "exponential", "gumbelmax"]
npar = [2, 2, 2, 3, 3, 4, 2, 2]

# Number of points to compare theoretical and empirical values
nS = 1000
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[i][:4], test, 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.

13.3.15.3 Graph

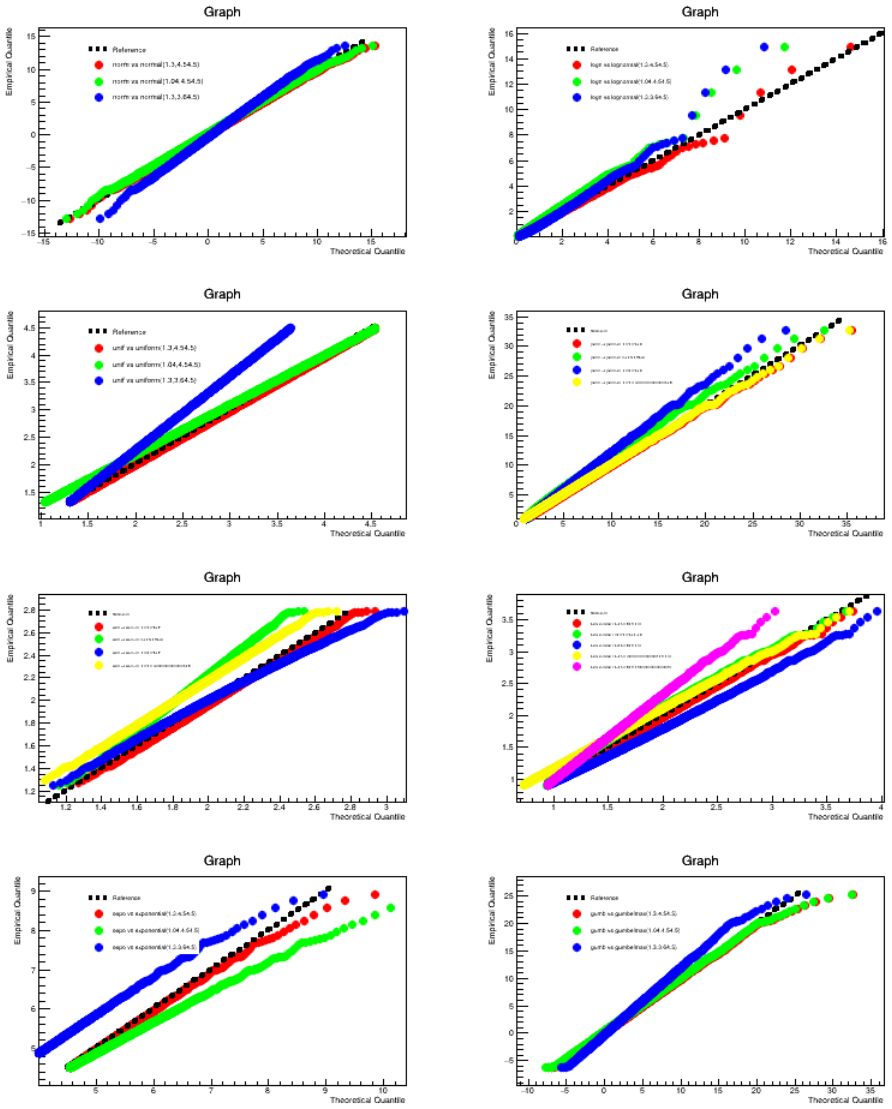


Figure 13.7: Graph of the macro "dataserverDrawQQPlot.py"

13.3.16 Macro "dataserverDrawPPPlot.py"

13.3.16.1 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.

13.3.16.2 Macro Uranie

```
"""
Example of PP plot
"""
```

(continues on next page)

(continued from previous page)

```
from URANIE import DataServer, Sampler
import ROOT

# Create a TDS with 8 kind of distributions
p1 = 1.3
p2 = 4.5
p3 = 0.9
p4 = 4.4  # Fixed values for parameters

tds0 = DataServer.TDataServer()
tds0.addAttribute(DataServer.TNormalDistribution("norm", p1, p2))
tds0.addAttribute(DataServer.TLogNormalDistribution("logn", p1, p2))
tds0.addAttribute(DataServer.TUniformDistribution("unif", p1, p2))
tds0.addAttribute(DataServer.TExponentialDistribution("expo", p1, p2))
tds0.addAttribute(DataServer.TGammaDistribution("gamm", p1, p2, p3))
tds0.addAttribute(DataServer.TBetaDistribution("beta", p1, p2, p3, p4))
tds0.addAttribute(DataServer.TWeibullDistribution("weib", p1, p2, p3))
tds0.addAttribute(DataServer.TGumbelMaxDistribution("gumb", p1, p2))

# Create the sample
fsamp = Sampler.TBasicSampling(tds0, "lhs", 200)
fsamp.generateSample()

# Define number of laws, their name and numbers of parameters
nLaws = 8
# number of parameters to put in () for the corresponding law
laws = ["normal", "lognormal", "uniform", "gamma",
        "weibull", "beta", "exponential", "gumbelmax"]
npar = [2, 2, 2, 3, 3, 4, 2, 2]

# Create the canvas
c = ROOT.TCanvas("c1", "", 800, 1000)
# Create the 8 pads
apad = ROOT.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
nS = 1000
mod = 0.8  # Factor used to artificially change the parameter values

Par = lambda i, n, p, mod: ","+str(p*mod) if n >= i else ""
opt = ""  # option of the drawPPPlot method
for i in range(nLaws):

    # Clean sstr
    test = ""
    # Add nominal configuration
    test += laws[i] + "(" + str(p1) + "," + str(p2) + Par(3, npar[i], p3, 1) \
        + str(p2) + Par(4, npar[i], p4, 1) + ")"
    # Changing par1
```

(continues on next page)

(continued from previous page)

```
    test += ":" + laws[i] + "(" + str(p1*mod) + "," + str(p2) \
        + Par(3, npar[i], p3, 1) + str(p2) + Par(4, npar[i], p4, 1) + ")"
    # Changing par2
    test += ":" + laws[i] + "(" + str(p1) + "," + str(p2*mod) \
        + Par(3, npar[i], p3, 1) + str(p2) + Par(4, npar[i], p4, 1) + ")"
    # Changing par3
    if npar[i] >= 3:
        test += ":" + laws[i] + "(" + str(p1) + "," + str(p2) \
            + Par(3, npar[i], p3, mod) + str(p2) + Par(4, npar[i], p4, 1) + ")"
    # Changing par4
    if npar[i] >= 4:
        test += ":" + laws[i] + "(" + str(p1) + "," + str(p2) \
            + Par(3, npar[i], p3, 1) + str(p2) + Par(4, npar[i], p4, mod) + ")"

    apad.cd(i+1)
    # Produce the plot
    tds0.drawPPPlot(laws[i][:4], test, nS, opt)
```

The macro is based on the one discussed in *Macro "dataserverDrawQQPlot.py"*. The only difference is this line

```
    tds0.drawPPPlot(laws[i][:4], test, 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.

13.3.16.3 Graph

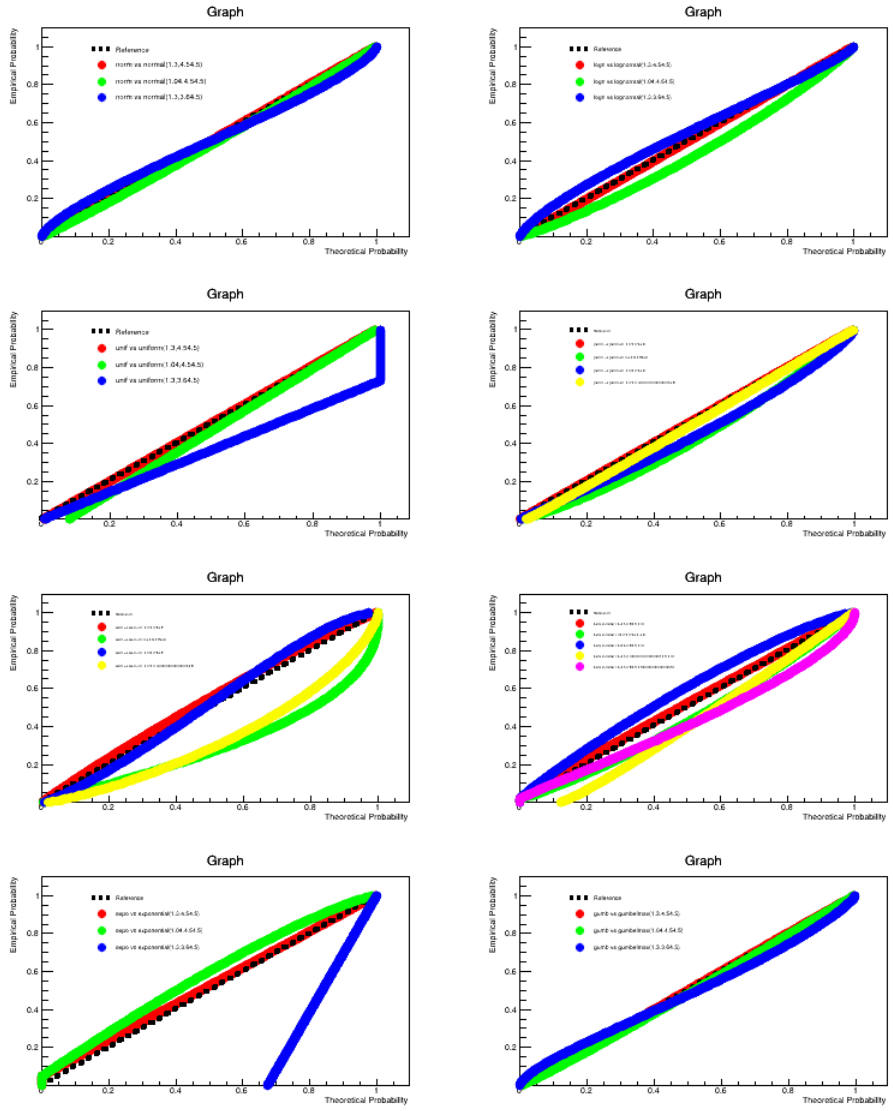


Figure 13.8: Graph of the macro "`dataserverDrawPPPlot.py`"

13.3.17 Macro “dataserverPCAExample.py”

13.3.17.1 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 *Combining these aspects: performing PCA*.

13.3.17.2 Macro Uranie

```
"""
Example of PCA usage (standalone case)
"""
```

(continues on next page)

(continued from previous page)

```
from URANIE import DataServer
import ROOT

# Read the database
tdsPCA = DataServer.TDataServer("tdsPCA", "my TDS")
tdsPCA.fileDataRead("Notes.dat")

# Create the PCA object precising the variables of interest
tpca = DataServer.TPCA(tdsPCA, "Maths:Physics:French:Latin:Music")
tpca.compute()

graphical = True  # do graphs
dumponscreen = True  # or dumping results
showcoordinate = False  # show the points coordinate while dumping results

if graphical:

    # Draw all point in PCA planes
    cPCA = ROOT.TCanvas("cpca", "PCA", 800, 800)
    apad1 = ROOT.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
    cLoading = ROOT.TCanvas("cLoading", "Loading Plot", 800, 800)
    apad2 = ROOT.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
    c = ROOT.TCanvas("cEigenValues", "Eigen Values Plot", 1100, 500)
    apad3 = ROOT.TPad("apad3", "apad3", 0, 0.03, 1, 1)
    apad3.Draw()
    apad3.cd()
    apad3.Divide(3, 1)
    ntd = tpca.getResultNtupleD()
    apad3.cd(1)
    ntd.Draw("eigen:i", "", "lp")
```

(continues on next page)

(continued from previous page)

```
    apad3.cd(2)
    ntd.Draw("eigen_pct:i", "", "lp")
    ROOT.gPad.SetGrid()
    apad3.cd(3)
    ntd.Draw("sum_eigen_pct:i", "", "lp")
    ROOT.gPad.SetGrid()

if dumponscreen:

    nPCused = 5  # 3 to see only the meaningful ones
    PCname = ""
    Cosname = ""
    Contrname = ""
    Variable = "Pupil"
    for iatt in range(1, nPCused+1):
        ##list of PC: PC_1:PC_2:PC_3:PC_4:PC_5
        PCname += "PC_"+str(iatt)+":"
        ##list of quality coeff: cosca_1:cosca_2:cosca_3:cosca_4:cosca_5
        Cosname += "cosca_"+str(iatt)+":"
        ##list of contribution: contr_1:contr_2:contr_3:contr_4:contr_5
        Contrname += "contr_"+str(iatt)+":"

    PCname = PCname[:-1]
    Cosname = Cosname[:-1]
    Contrname = Contrname[:-1]

    print("\n====== EigenValues ======")
    tpca.getResultNtupleD().Scan("*")

    if showcoordinate:
        print("\n====== EigenVectors ======")
        tpca._matEigenVectors.Print()

        print("\n====== New Coordinates ======")
        tdsPCA.scan((Variable+":"+PCname))

    varRes = tpca.getVariableResultNtupleD()
    print("\n====== Looking at variables: Quality of representation ======")
    varRes.Scan(("Variable:"+Cosname))

    print("\n====== Looking at variables: Contribution to axis ======")
    varRes.Scan(("Variable:"+Contrname))

    print("\n====== Looking at events:  Quality of representation ======")
    tdsPCA.scan((Variable+":"+Cosname))

    print("\n====== Looking at events: Contribution to axis ========")
    tdsPCA.scan((Variable+":"+Contrname))
```

This first part of this macro is described in *Combining these aspects: performing PCA*. 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

```
    nPCused = 5  # 3 to see only the meaningful ones
    PCname = ""
    Cosname = ""
    Contrname = ""
    Variable = "Pupil"
    for iatt in range(1, nPCused+1):
        ##list of PC: PC_1:PC_2:PC_3:PC_4:PC_5
        PCname += "PC_"+str(iatt)+":"
        ##list of quality coeff: cosca_1:cosca_2:cosca_3:cosca_4:cosca_5
        Cosname += "cosca_"+str(iatt)+":"
        ##list of contribution: contr_1:contr_2:contr_3:contr_4:contr_5
        Contrname += "contr_"+str(iatt)+":"

    PCname = PCname[:-1]
    Cosname = Cosname[:-1]
    Contrname = Contrname[:-1]
```

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 *Console*).

```
    varRes = tpca.getVariableResultNtupleD()
    print("\n====== Looking at variables: Quality of representation ======")
    varRes.Scan(("Variable:"+Cosname))

    print("\n====== Looking at variables: Contribution to axis ======")
    varRes.Scan(("Variable:"+Contrname))
```

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 *Console*).

```
    print("\n====== Looking at events:  Quality of representation ======")
    tdsPCA.scan((Variable+":"+Cosname))

    print("\n====== Looking at events: Contribution to axis ========")
    tdsPCA.scan((Variable+":"+Contrname))
```

13.3.17.3 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 *
```

(continues on next page)

(continued from previous page)

```
*        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 *
```

(continues on next page)

(continued from previous page)

```
*        8 *   Evelyne * 0.0150734 * 0.2222843 * 0.4479453 * 0.0864396 * 0.0865925 *
*****************************************************************************
```

13.4 Macros Sampler

13.4.1 Macro “samplingFlowrate.py”

13.4.1.1 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.

13.4.1.2 Macro Uranie

```
"""
Example of simple flowrate DoE generation
"""
from URANIE import DataServer, Sampler
import ROOT

nS = 1000
# Define the DataServer
tds = DataServer.TDataServer("tdsFlowrate", "Design of experiments")

# Add the study attributes
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))


# test Generateur de plan d'experience
sampling = Sampler.TSampling(tds, "lhs", nS)
sampling.generateSample()

tds.exportData("_flowrate_sampler_.dat")


# Visualisation
Canvas = ROOT.TCanvas("Canvas", "Graph samplingFlowrate", 5, 64, 1270, 667)
ROOT.gStyle.SetPalette(1)
pad = ROOT.TPad("pad", "pad", 0, 0.03, 1, 1)
pad.Draw()

pad.Divide(2, 2)
pad.cd(1)
tds.draw("r")
```

(continues on next page)

(continued from previous page)

```
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(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))
```

The sampling is generated with the LHS method:

```
sampling = Sampler.TSampling(tds, "lhs", nS)
sampling.generateSample()
```

Data are exported in an ASCII file:

```
tds.exportData("_flowrate_sampler_.dat")
```

13.4.1.3 Graph

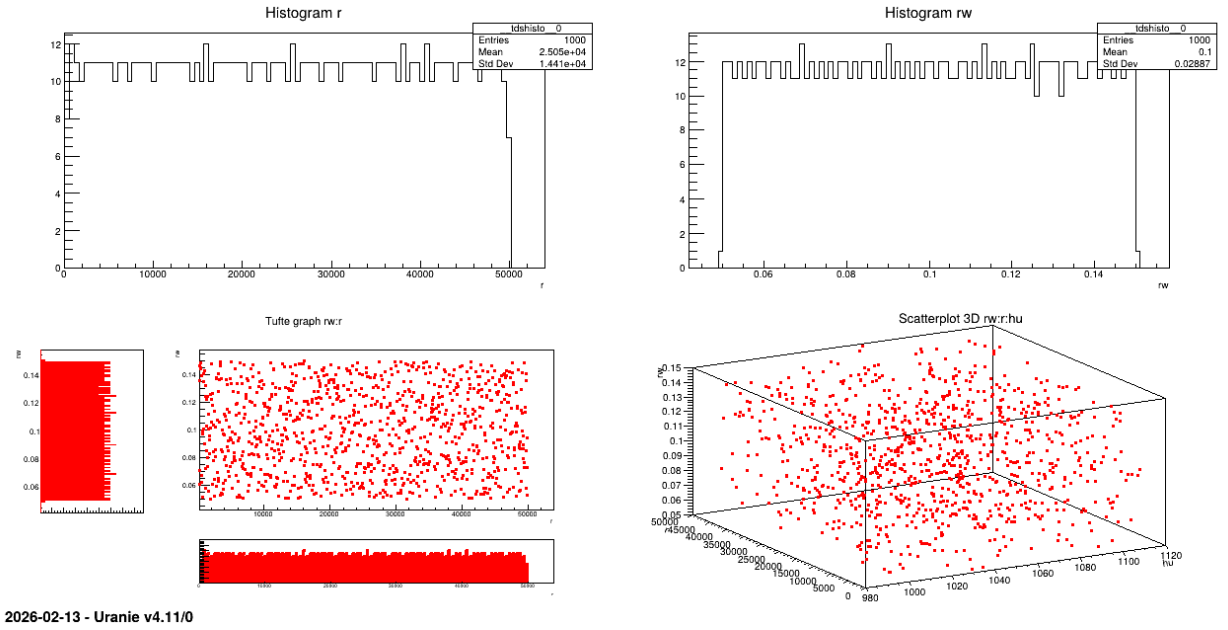


Figure 13.9: Graph of the macro **"samplingFlowrate.py"**

13.4.2 Macro “samplingLHS.py”

13.4.2.1 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.

13.4.2.2 Macro Uranie

```
"""
Example of LHS DoE sampling
"""
from URANIE import DataServer, Sampler
import ROOT

# Create a DataServer.TDataServer
tds = DataServer.TDataServer()
# Fill the DataServer with the three attributes of the study
tds.addAttribute(DataServer.TUniformDistribution("x1", 3., 4.))
tds.addAttribute(DataServer.TNormalDistribution("x2", 0.5, 1.5))
tds.addAttribute(DataServer.TTriangularDistribution("x3", 1., 5., 4.))


# Generate the sampling from the DataServer.TDataServer
sampling = Sampler.TSampling(tds, "lhs", 5000)
sampling.generateSample()

tds.StartViewer()

# Graph
Canvas = ROOT.TCanvas("c1", "Graph for the Macro sampling", 5, 64, 1270, 667)
pad = ROOT.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(DataServer.TUniformDistribution("x1", 3., 4.))
tds.addAttribute(DataServer.TNormalDistribution("x2", 0.5, 1.5))
tds.addAttribute(DataServer.TTriangularDistribution("x3", 1., 5., 4.))
```

The sampling is generated with the LHS method!

```
sampling = Sampler.TSampling(tds, "lhs", 5000)
sampling.generateSample()
```

13.4.2.3 Graph

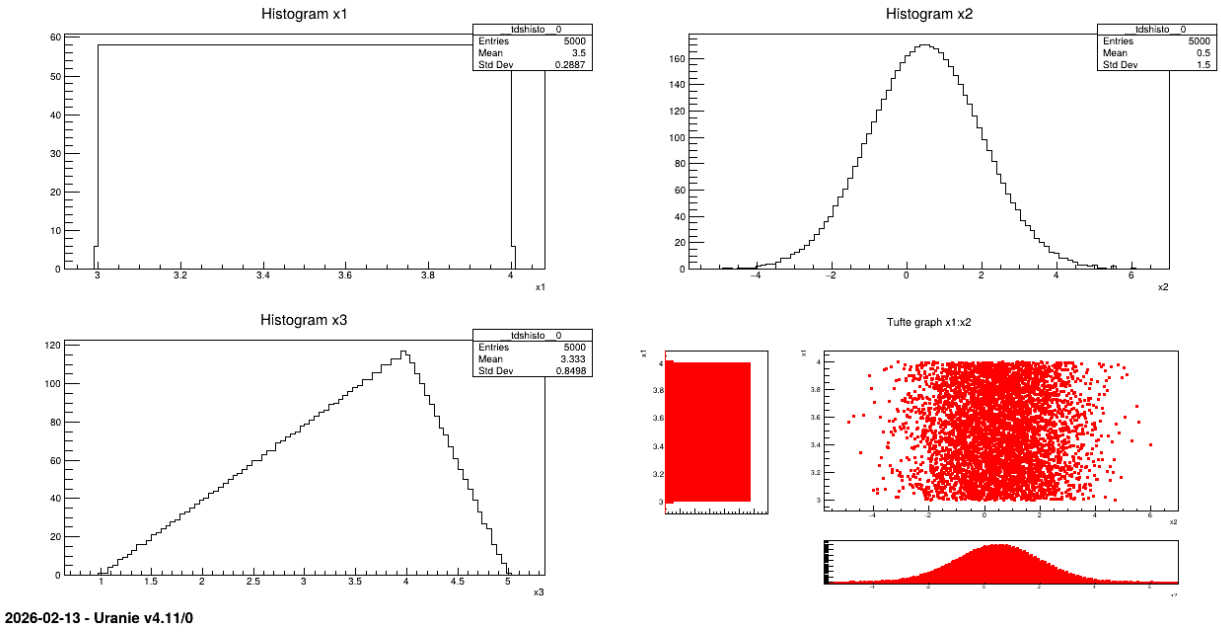


Figure 13.10: Graph of the macro **"samplingLHS.py"**

13.4.3 Macro "samplingLHSCorrelation.py"

13.4.3.1 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.* *Description of a correlation* and in [Bla17] 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.

13.4.3.2 Macro Uranie

```
"""
Example of simple LHS correlated DoE generation
"""
from URANIE import DataServer, Sampler
import ROOT

# Create a DataServer.TDataServer
tds = DataServer.TDataServer()
```

(continues on next page)

(continued from previous page)

```
# Fill the DataServer with the three attributes of the study
tds.addAttribute(DataServer.TUniformDistribution("x1", 3., 4.))
tds.addAttribute(DataServer.TNormalDistribution("x2", 0.5, 1.5))
tds.addAttribute(DataServer.TTriangularDistribution("x3", 1., 5., 4.))

# Generate the sampling from the DataServer.TDataServer
sampling = Sampler.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
Canvas = ROOT.TCanvas("c1", "Graph samplingCorrelation", 5, 64, 1270, 667)
pad = ROOT.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(DataServer.TUniformDistribution("x1", 3., 4.))
tds.addAttribute(DataServer.TNormalDistribution("x2", 0.5, 1.5))
tds.addAttribute(DataServer.TTriangularDistribution("x3", 1., 5., 4.))
```

The sampling is generated with the LHS method and a correlation is set between the two first attributes:

```
sampling = Sampler.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")
```

13.4.3.3 Graph

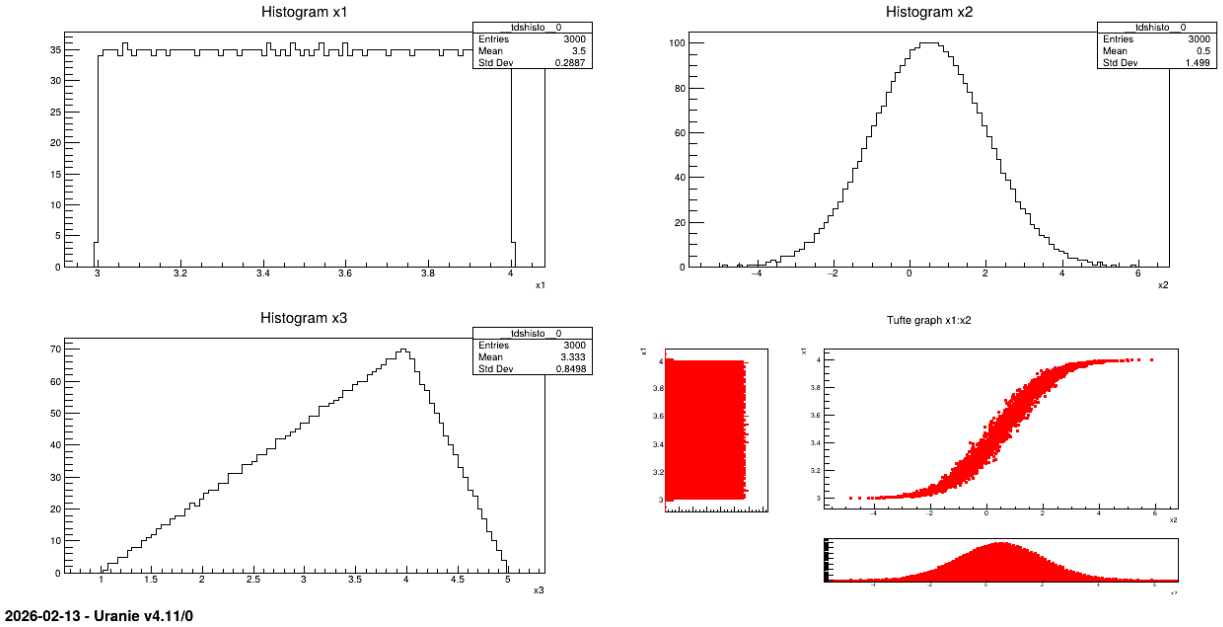


Figure 13.11: Graph de la macro **"samplingLHSCorrelation.py"**

13.4.4 Macro "samplingQMC.py"

13.4.4.1 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.

13.4.4.2 Macro Uranie

```
"""
Example of quasi monte-carlo generation
"""
from URANIE import DataServer, Sampler
import ROOT

# Parameters
nSampler = 100
sQMC = "halton"  # halton / sobol
nVar = 12

# Create a DataServer.TDataServer
tds = DataServer.TDataServer()

# Fill the DataServer with the nVar attributes of the study
for i in range(nVar):
    tds.addAttribute(DataServer.TUniformDistribution("x"+str(i+1), -3.0, 2.0))
```

(continues on next page)

(continued from previous page)

```
# Generate the quasi Monte-Carlo sequence from the DataServer.TDataServer
qmc = Sampler.TQMC(tds, sQMC, nSampler)
qmc.generateSample()

# Visualisation
Canvas = ROOT.TCanvas("c1", "Graph for the Macro qmc", 5, 64, 1270, 667)
Canvas.Range(0, 0, 25, 18)

pl = ROOT.TPaveLabel(1, 16.3, 24, 17.5, "qMC sequence : "+sQMC, "br")
pl.SetBorderSize(0)
pl.Draw()

pad1 = ROOT.TPad("pad1", "Determ", 0.02, 0.05, 0.48, 0.88)
pad2 = ROOT.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 i in range(nVar):
    tds.addAttribute(DataServer.TUniformDistribution("x"+str(i+1), -3.0, 2.0))
```

The sampling is generated with the QMC method and a correlation is set between the two first attributes:

```
qmc = Sampler.TQMC(tds, sQMC, nSampler)
qmc.generateSample()
```

The rest of the code is used to produce a plot.

13.4.4.3 Graph

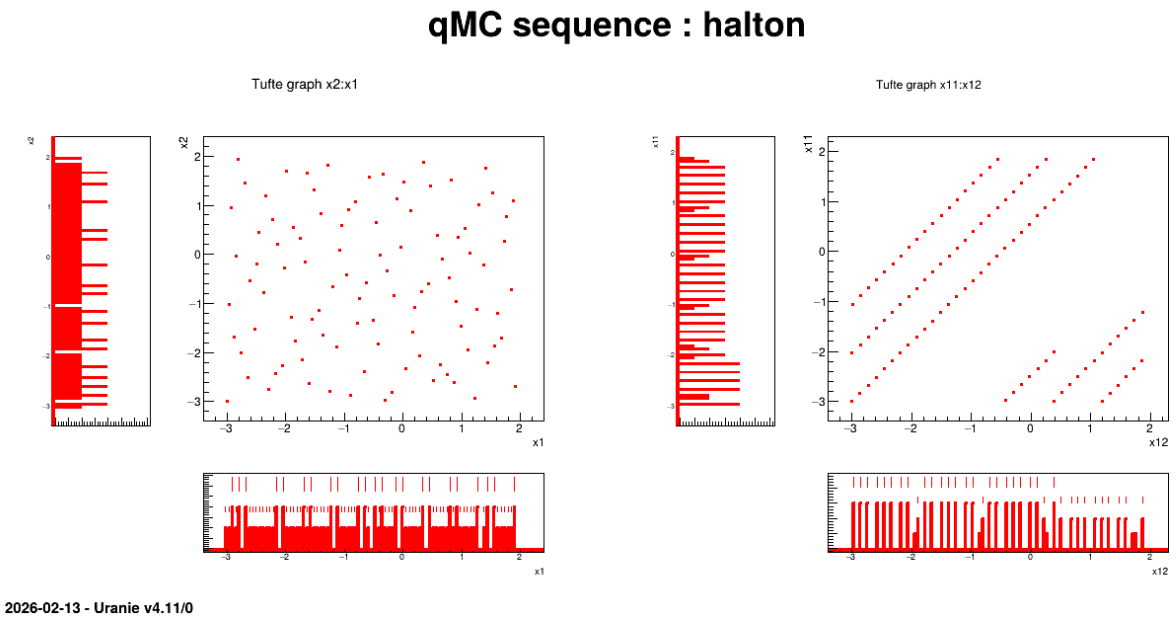


Figure 13.12: Graph of the macro **"samplingQMC.py"**

13.4.5 Macro "samplingBasicSampling.py"

13.4.5.1 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.

13.4.5.2 Macro Uranie

```
"""
Example of simple DoE generations
"""
from URANIE import DataServer, Sampler
import ROOT

def test_big_sampling():
    """Produce a large sample."""
    tds = DataServer.TDataServer()
    print("    - Creating attributes...")
    for i in range(4000):
        tds.addAttribute(DataServer.TUniformDistribution("x_"+str(i), 0., 1.))
    sam = Sampler.TBasicSampling(tds, "srs", 5000)
    print("    - Starting sampling...")
    sam.generateSample()
```

(continues on next page)

(continued from previous page)

```
def test_small_srs_sampling():
    """Produce a small srs sample."""
    tds = DataServer.TDataServer()
    print("    - Creating attributes...")
    for i in range(10000):
        tds.addAttribute(DataServer.TUniformDistribution("x_"+str(i), 0., 1.))
    sam = Sampler.TBasicSampling(tds, "srs", 10)
    print("    - Starting sampling...")
    sam.generateSample()


def test_small_lhs_sampling():
    """Produce a small lhs sample."""
    tds = DataServer.TDataServer()
    print("    - Creating attributes...")
    for i in range(10000):
        tds.addAttribute(DataServer.TUniformDistribution("x_"+str(i), 0., 1.))
    sam = Sampler.TBasicSampling(tds, "lhs", 10)
    print("    - Starting sampling...")
    sam.generateSample()


print("*****************************************************")
print(" ** Test small SRS sampling (10000 attributes, 10 data)")
test_small_srs_sampling()
print("*****************************************************")
ROOT.gROOT.ls()

print("*****************************************************")
print(" ** Test small LHS sampling (10000 attributes, 10 data)")
test_small_lhs_sampling()
print("*****************************************************")
ROOT.gROOT.ls()

print("*****************************************************")
print(" ** Test big sampling (4000 attributes, 5000 data)")
test_big_sampling()
print("*****************************************************")
ROOT.gROOT.ls()
```

13.4.6 Macro “samplingOATRegular.py”

13.4.6.1 Objective

This part shows the complete code used to produce the console display in *Regular mode*.

13.4.6.2 Macro Uranie

```
"""
Example of regular OAT generation
"""
from URANIE import DataServer, Sampler
```

(continues on next page)

(continued from previous page)

```
# step 1
tds = DataServer.TDataServer("tdsoat", "Data server for simple OAT design")
tds.addAttribute(DataServer.TAttribute("x1"))
tds.addAttribute(DataServer.TAttribute("x2"))

# step 2
tds.getAttribute("x1").setDefaultValue(0.0)
tds.getAttribute("x2").setDefaultValue(10.0)

# step 3
oatSampler = Sampler.TOATDesign(tds, "regular", 4)

# step 4
use_percentage = True
oatSampler.setRange("x1", 2.0)
oatSampler.setRange("x2", 40.0, use_percentage)

# step 5
oatSampler.generateSample()

# display
tds.scan("*", "", "colsize=15 col=:4:3:")
```

13.4.7 Macro “samplingOATRandom.py”

13.4.7.1 Objective

This part shows the complete code used to produce the console display in *Random mode*.

13.4.7.2 Macro Uranie

```
"""
Example of OAT generation with random locations
"""
from URANIE import DataServer, Sampler
import ROOT

# step 1
tds = DataServer.TDataServer("tdsoat", "Data server for simple OAT design")
tds.addAttribute(DataServer.TUniformDistribution("x1", -5.0, 5.0))
tds.addAttribute(DataServer.TNormalDistribution("x2", 11.0, 1.0))

# step 2
tds.getAttribute("x1").setDefaultValue(0.0)
tds.getAttribute("x2").setDefaultValue(10.0)

# step 3
oatSampler = Sampler.TOATDesign(tds, "lhs", 1000)

# step 4
use_percentage = True
```

(continues on next page)

(continued from previous page)

```
oatSampler.setRange("x1", 2.0)
oatSampler.setRange("x2", 40.0, use_percentage)

# step 5
oatSampler.generateSample()

# display
c = ROOT.TCanvas("can", "can", 10, 32, 1200, 600)
apad = ROOT.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")
```

13.4.8 Macro “samplingOATMulti.py”

13.4.8.1 Objective

This part shows the complete code used to produce the console display in *Multiple sets of nominal values*.

13.4.8.2 Macro Uranie

```
"""
Example of OAT multi start generation
"""
from URANIE import DataServer, Sampler

# step 1
tds = DataServer.TDataServer("tdsoat", "Data server for simple OAT design")
tds.fileDataRead("myNominalValues.dat")

# step 3
oatSampler = Sampler.TOATDesign(tds)

# step 4
use_percentage = True
oatSampler.setRange("x1", 2.0)
oatSampler.setRange("x2", 40.0, use_percentage)

# step 5
oatSampler.generateSample()

# display
tds.scan("tdsoat__n__iter__:x1:x2:__nominal_set__:__modified_att__", "", "colsize=15␣
↪col=:4:4:")
```

13.4.9 Macro “samplingOATRange.py”

13.4.9.1 Objective

This part shows the complete code used to produce the console display in *Multiple ranges*.

13.4.9.2 Macro Uranie

```
"""
Example of OAT generation with range options
"""
from URANIE import DataServer, Sampler

# step 1
tds = DataServer.TDataServer("tds", "Data server for simple OAT design")
tds.fileDataRead("myNominalValues.dat")

# step 3
oatSampler = Sampler.TOATDesign(tds)

# step 4
use_percentage = True
oatSampler.setRange("x1", "rx1")
oatSampler.setRange("x2", 40.0, use_percentage)

# step 5
oatSampler.generateSample()

# display
tds.scan("tds__n__iter__:x1:x2:rx1:__nominal_set__:__modified_att__", "", "colsize=12␣
↪col=:4:4:4:15:15:")
```

13.4.10 Macro “samplingSpaceFilling.py”

13.4.10.1 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.

13.4.10.2 Macro Uranie

```
"""
Example of space filling DoE
"""
from URANIE import DataServer, Sampler
import ROOT

def generate_and_draw_it(l_pad, l_tds, l_tsp, l_nb, l_title):
    """Delete the tuple, generate a sample and draw the plot."""

    l_pad.cd(l_nb+1)
    l_tds.deleteTuple()
    l_tsp.generateSample()
    l_tds.drawTufte("x2:x1")
    l_pad.GetPad(l_nb+1).GetPrimitive("TPave").SetLabel("")
    ROOT.gPad.GetPrimitive("htemp").SetTitle(l_title)
    ROOT.gPad.Modified()

# Attributes
att1 = DataServer.TUniformDistribution("x1", 10, 12)
att2 = DataServer.TUniformDistribution("x2", 0, 3)
nor1 = DataServer.TNormalDistribution("x1", 0, 1)
nor2 = DataServer.TNormalDistribution("x2", 3, 5)

# Pointer to DataServer and Samplers
tds = [0., 0., 0.]
tsp = [0., 0., 0., 0., 0., 0., 0., 0., 0.]
algoname = ["SaltelliA", "SaltelliB", "Cukier"]

# Canvas to produce the 3x3 plot
Can = ROOT.TCanvas("Can", "Can", 10, 32, 1200, 1200)
pad = ROOT.TPad("pad", "pad", 0, 0.03, 1, 1)
pad.Draw()
pad.Divide(3, 3)
ct = 0

for itds in range(3):

    # Create a DataServer to store new configuration
    # (UnivsUni, GausvsUni, Gaus vs Gaus)
    tds[itds] = DataServer.TDataServer("test", "test")
    if itds == 0:
        tds[itds].addAttribute(att1)
        tds[itds].addAttribute(att2)
    if itds == 1:
        tds[itds].addAttribute(att1)
        tds[itds].addAttribute(nor2)
    if itds == 2:
        tds[itds].addAttribute(nor1)
        tds[itds].addAttribute(nor2)

    # Looping over the 3 spacefilling algo available
```

(continues on next page)

(continued from previous page)

```
    for ialg in range(3):
        # Instantiate the sampler
        if ialg == 0:
            tsp[ct] = Sampler.TSpaceFilling(tds[itds], "srs", 1000,
                                            Sampler.TSpaceFilling.kSaltelliA)
        if ialg == 1:
            tsp[ct] = Sampler.TSpaceFilling(tds[itds], "srs", 1000,
                                            Sampler.TSpaceFilling.kSaltelliB)
        if ialg == 2:
            tsp[ct] = Sampler.TSpaceFilling(tds[itds], "srs", 1000,
                                            Sampler.TSpaceFilling.kCukier)

        # Draw with correct legend
        generate_and_draw_it(pad, tds[itds], tsp[ct], ct, algoname[ialg])
        ct = ct+1
```

13.4.10.3 Graph

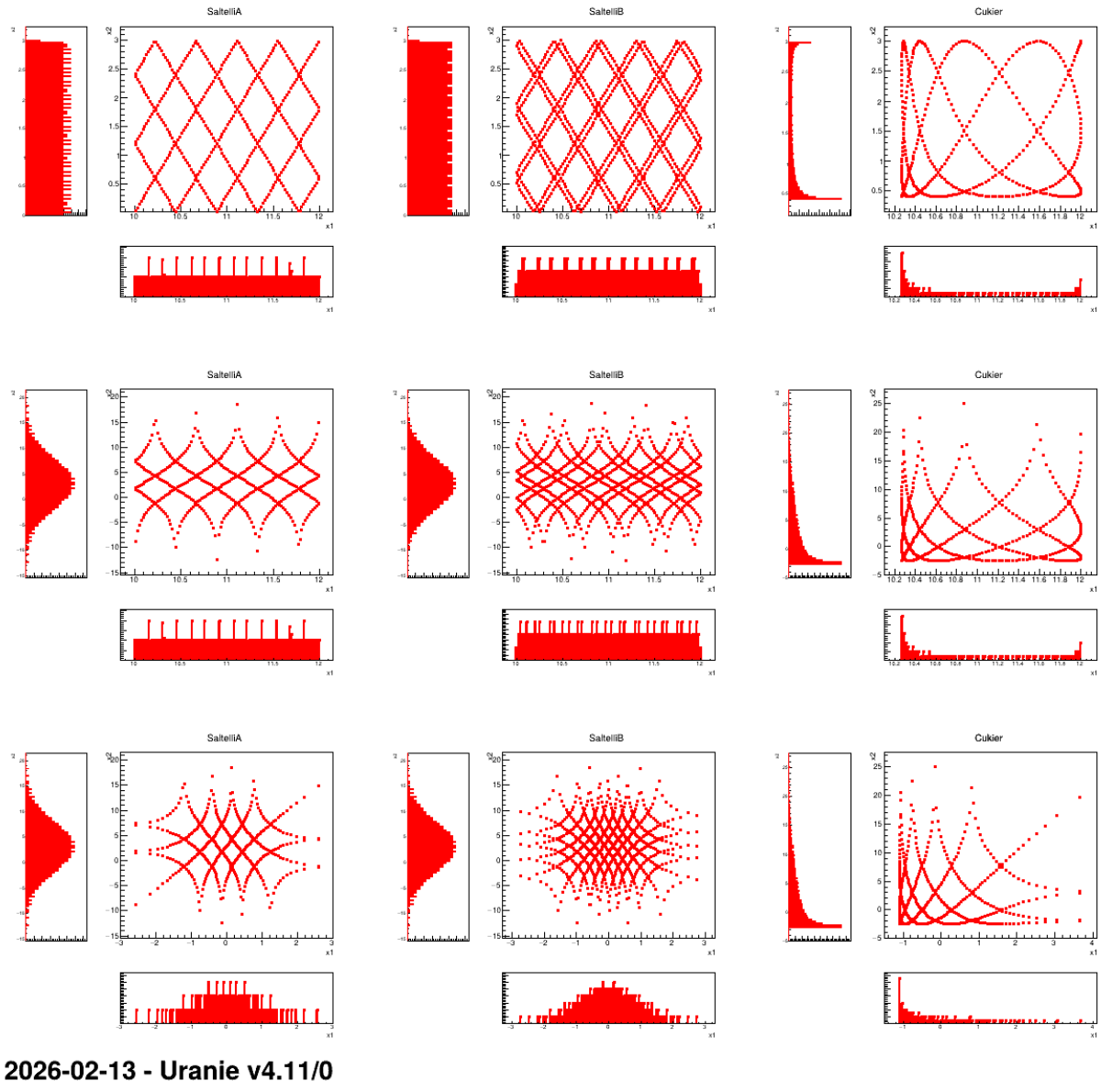


Figure 13.13: Graph of the macro **"samplingSpaceFilling.py"**

13.4.11 Macro "samplingMaxiMinLHSFromLHSGrid.py"

13.4.11.1 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.

13.4.11.2 Macro Uranie

```
"""
Example of maximin lhs sample generation
"""
from URANIE import DataServer, Sampler
import ROOT

def niceplot(l_tds, l_lat, l_title):
    """Generate the grid plot."""

    l_tds.getTuple().SetMarkerStyle(20)
    l_tds.getTuple().SetMarkerSize(1)
    l_tds.Draw("X2:X1")
    tit = "MinDist = "+str(Sampler.TMaxiMinLHS.getMinDist(l_tds.getMatrix()))
    ROOT.gPad.GetPrimitive("__tdshisto__0").SetTitle("")
    ROOT.gPad.GetPrimitive("__tdshisto__0").GetXaxis().SetRangeUser(0., 1.)
    ROOT.gPad.GetPrimitive("__tdshisto__0").GetYaxis().SetRangeUser(0., 1.)
    l_lat.DrawLatex(0.25, 0.94, l_title+", "+tit)

# Canvas to produce the 2x1 plot to compare LHS designs
Can = ROOT.TCanvas("Can", "Can", 10, 32, 1200, 550)
pad = ROOT.TPad("pad", "pad", 0, 0.03, 1, 1)
pad.Draw()
pad.Divide(2, 1)

size = 20  # Size of the samples to be produced
lat = ROOT.TLatex()
lat.SetNDC()
lat.SetTextSize(0.038)  # To write titles

# Create dataserver and define the attributes
tds = DataServer.TDataServer("tds", "pouet")
tds.addAttribute(DataServer.TUniformDistribution("X1", 0, 1))
tds.getAttribute("X1").delShare()  # state to tds that it owns it
tds.addAttribute(DataServer.TUniformDistribution("X2", 0, 1))
tds.getAttribute("X2").delShare()  # state to tds that it owns it

# Generate the original LHS grid
sampl = Sampler.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
maxim = Sampler.TMaxiMinLHS(tds, size, 0.1, 0.99, 300, 300)
maxim.generateSample()
```

(continues on next page)

(continued from previous page)

```
# 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
maxim = Sampler.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 [Bla17]

13.4.11.3 Graph

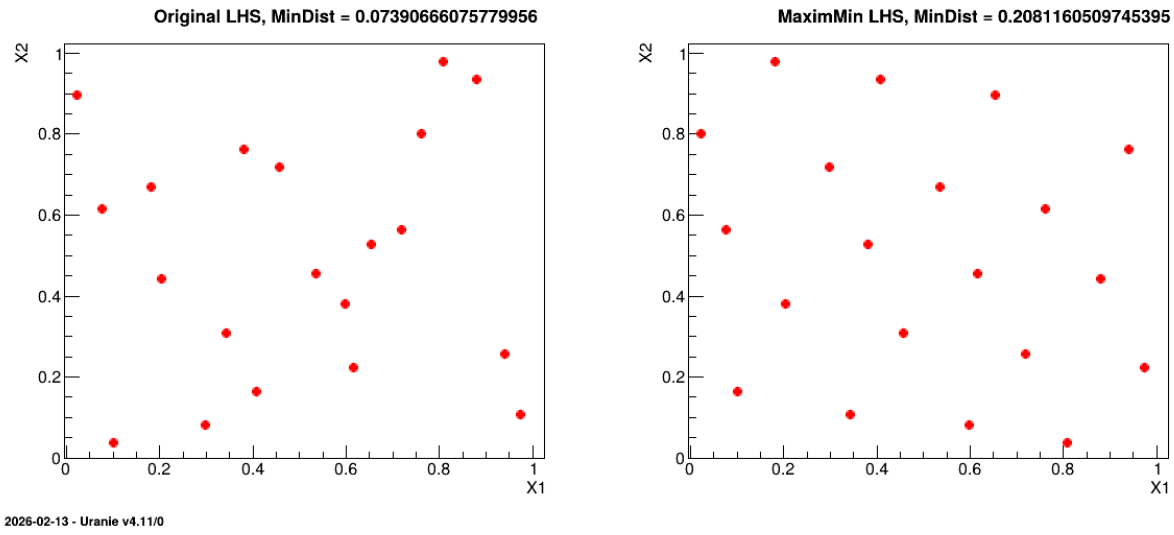


Figure 13.14: Graph of the macro **"samplingMaxiMinLHSFromLHSGrid.py"**

13.4.12 Macro "samplingConstrLHSLinear.py"

13.4.12.1 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).

13.4.12.2 Macro Uranie

```
"""
Example of constrained lhs with linear constraints
"""
import numpy as np
from URANIE import DataServer, Sampler
import ROOT

def nicegrid(l_tds, l_can):
    """Generate the grid plot."""

    l_ns = l_tds.getNPatterns()
    l_nx = l_tds.getNAttributes()
    l_tds.drawPairs()

    all_h = []
    ROOT.gStyle.SetOptStat(0)
    for iatt in range(l_nx):

        l_can.GetPad(iatt*(l_nx+1)+1).cd()
        att = l_tds.getAttribute(iatt)
        a_name = att.GetName()
        all_h.append(ROOT.TH1F(a_name+"_histo", a_name+";x_"+str(iatt),
                               l_ns, att.getLowerBound(), att.getUpperBound()))
        l_tds.Draw(a_name+">>"+a_name+"_histo", "", "goff")
        all_h[iatt].Draw()

    return all_h

# Canvas to produce the pair plot
Can = ROOT.TCanvas("Can", "Can", 10, 32, 1400, 1000)
apad = ROOT.TPad("apad", "apad", 0, 0.03, 1, 1)
apad.Draw()
apad.cd()

ns = 250  # Size of the samples to be produced
nx = 3

# Create dataserver and define the attributes
tds = DataServer.TDataServer("tds", "pouet")
for i in range(nx):

    tds.addAttribute(DataServer.TUniformDistribution("x_"+str(i), 0, i+1))
    tds.getAttribute("x_"+str(i)).delShare()

ROOT.gROOT.LoadMacro("ConstrFunctions.C")

# Generate the constr lhs
constrlhs = Sampler.TConstrLHS(tds, ns)
inputs = np.array([1, 0, 2, 1], dtype='i')
constrlhs.addConstraint(ROOT.Linear, 2, len(inputs), inputs)
constrlhs.generateSample()
```

(continues on next page)

(continued from previous page)

```
# Do the plot
ListOfDiag = nicegrid(tds, apad)
```

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
constrlhs = Sampler.TConstrLHS(tds, ns)
inputs = np.array([1, 0, 2, 1], dtype='i')
constrlhs.addConstraint(ROOT.Linear, 2, len(inputs), inputs)
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 *TConstrLHS example*. The first argument of this method is the pointer to the C++ function which has been included in our macro through a line juste before the sampler creation:

```
ROOT.gROOT.LoadMacro("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 13.15

13.4.12.3 Graph

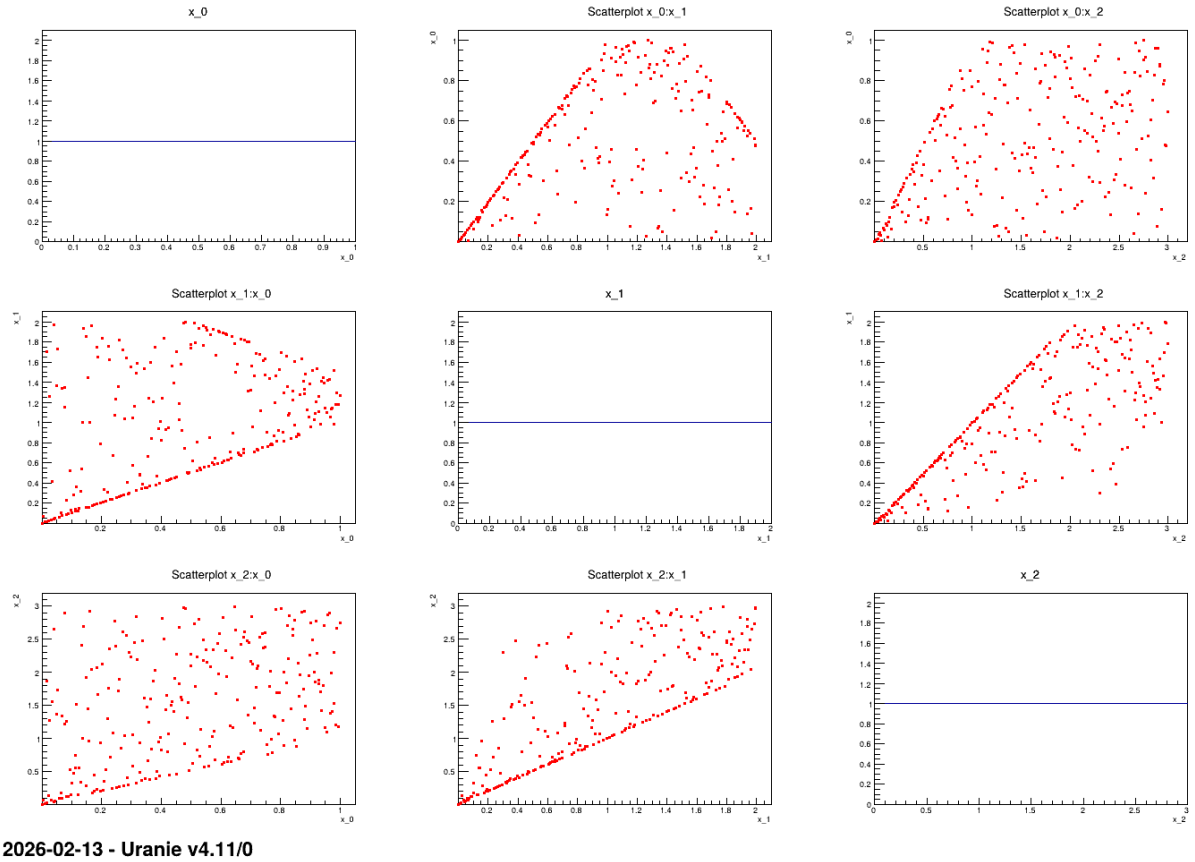


Figure 13.15: Graph of the macro **"samplingConstrLHSLinear.py"**

13.4.13 Macro "samplingConstrLHSEllipses.py"

13.4.13.1 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.

13.4.13.2 Macro Uranie

```
"""
Example of Constrained lhs with ellipses
"""
import numpy as np
from URANIE import DataServer, Sampler
import ROOT

def nicegrid(l_tds, l_can):
    """Generate the grid plot."""
```

(continues on next page)

(continued from previous page)

```
    l_ns = l_tds.getNPatterns()
    l_nx = l_tds.getNAttributes()
    l_tds.drawPairs()

    all_h = []
    ROOT.gStyle.SetOptStat(0)
    for iatt in range(l_nx):
        l_can.GetPad(iatt*(l_nx+1)+1).cd()
        att = l_tds.getAttribute(iatt)
        a_name = att.GetName()
        all_h.append(ROOT.TH1F(a_name+"_histo", a_name+";x_"+str(iatt),
                               l_ns, att.getLowerBound(), att.getUpperBound()))
        l_tds.Draw(a_name+">>"+a_name+"_histo", "", "goff")
        all_h[iatt].Draw()

    return all_h

# Canvas to produce the pair plot
Can = ROOT.TCanvas("Can", "Can", 10, 32, 1400, 1000)
apad = ROOT.TPad("apad", "apad", 0, 0.03, 1, 1)
apad.Draw()
apad.cd()

ns = 250  # Size of the samples to be produced
nx = 3

# Create dataserver and define the attributes
tds = DataServer.TDataServer("tds", "pouet")
for i in range(nx):

    tds.addAttribute(DataServer.TUniformDistribution("x_"+str(i), 0, i+1))
    tds.getAttribute("x_"+str(i)).delShare()

ROOT.gROOT.LoadMacro("ConstrFunctions.C")

# Generate the constr lhs
constrlhs = Sampler.TConstrLHS(tds, ns)
inputs = np.array([1, 0, 1, 2], dtype='i')
constrlhs.addConstraint(ROOT.CircularRules, 2, len(inputs), inputs)
constrlhs.generateSample()

# Do the plot
ListOfDiag = nicegrid(tds, apad)
```

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
constrlhs = Sampler.TConstrLHS(tds, ns)
inputs = np.array([1, 0, 1, 2], dtype='i')
constrlhs.addConstraint(ROOT.CircularRules, 2, len(inputs), inputs)
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 *TConstrLHS example*. The first argument of this method is the pointer to the C++ function which has been included in our macro through the line used right before the sampler creation:

```
ROOT.gROOT.LoadMacro("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 13.16

13.4.13.3 Graph

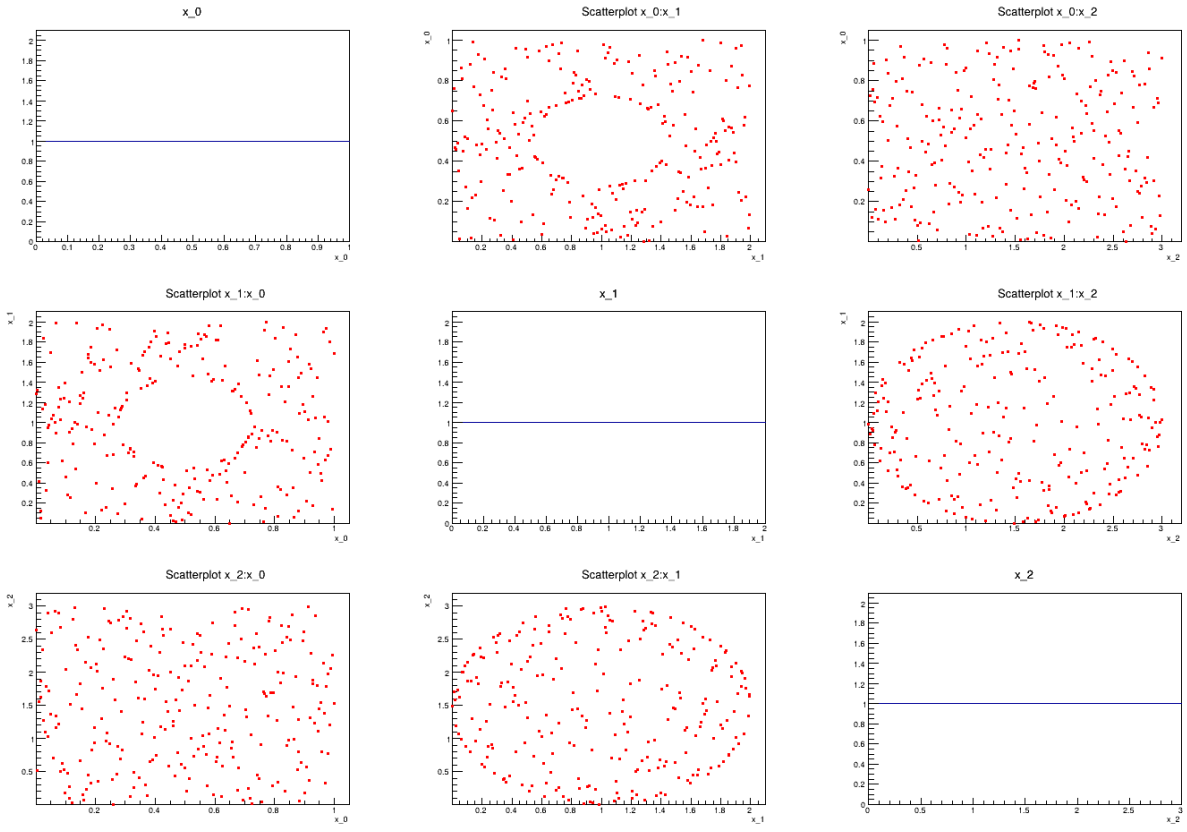


Figure 13.16: Graph of the macro **"samplingConstrLHSEllipses.py"**

13.4.14 Macro “samplingSingularCorrelationCase.py”

13.4.14.1 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.

13.4.14.2 Macro Uranie

```
"""
Example of DoE generation when the correlation matrix is singular
"""
from math import sqrt
from URANIE import DataServer, Sampler
import ROOT

# What classes to be used (true==TSampling, false==TBasicSampling)
ImanConover = False
SamplingType = "srs"
# Remove svd to use Cholesky
SamplerOption = "svd"

# Generating randomly a singular correlation matrix
# dimension of the  problem in total.
n = 10
# Number of dimension self explained.
# SHOULD BE SMALLER (singular) OR EQUAL TO (full-rank) n.
p = 6
# number of location in the doe
m = 300

# ============================================================================
#           Internal recipe to get this correlation matrix
# ============================================================================
A = ROOT.TMatrixD(n, p)
Rand = ROOT.TRandom3()
for i in range(n):
    for j in range(p):
        A[i][j] = Rand.Gaus(0., 1.)

Gamma = ROOT.TMatrixD(n, n)
Gamma.MultT(A, A)
Gamma *= 1./n

Sig = ROOT.TMatrixD(n, n)
for i in range(n):
    Sig[i][i] = 1./sqrt(Gamma[i][i])

Corr = ROOT.TMatrixD(Sig, ROOT.TMatrix.kMult, Gamma)
Corr *= Sig
```

(continues on next page)

(continued from previous page)

```
# ========================================================================
#                    Corr is our correlation matrix.
# ========================================================================


# Creating the TDS
tds = DataServer.TDataServer("pouet", "pouet")

# Adding attributes
for i in range(n):
    tds.addAttribute(DataServer.TNormalDistribution("n"+str(i), 0., 1.))

# Create the sampler and generate the doe
if ImanConover:
    sam = Sampler.TSampling(tds, SamplingType, m)
    sam.setCorrelationMatrix(Corr)
    sam.generateSample(SamplerOption)
else:
    sam = Sampler.TBasicSampling(tds, SamplingType, m)
    sam.setCorrelationMatrix(Corr)
    sam.generateCorrSample(SamplerOption)

# Compute the empirical correlation matrix
ResultCorr = tds.computeCorrelationMatrix()
# Change it into a residual matrix
ResultCorr -= Corr

ROOT.gStyle.SetOptStat(1110)
# Plot the results
c = ROOT.TCanvas("c", "c", 1800, 900)
apad = ROOT.TPad("apad", "apad", 0, 0.03, 1, 1)
apad.Draw()
apad.cd()
apad.Divide(2, 1)
apad.cd(1)
# Residual distribution
h = ROOT.TH1F("h", ";#Delta_{#rho}", 100, -0.2, 0.2)
for i in range(n):
    for j in range(i, n):
        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 [Bla17] 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)
ImanConover = False
SamplingType = "srs"
# Remove svd to use Cholesky
SamplerOption = "svd"

# Generating randomly a singular correlation matrix
# dimension of the  problem in total.
n = 10
# Number of dimension self explained.
# SHOULD BE SMALLER (singular) OR EQUAL TO (full-rank) n.
p = 6
# number of location in the doe
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:
    sam = Sampler.TSampling(tds, SamplingType, m)
    sam.setCorrelationMatrix(Corr)
    sam.generateSample(SamplerOption)
else:
    sam = Sampler.TBasicSampling(tds, SamplingType, m)
    sam.setCorrelationMatrix(Corr)
    sam.generateCorrSample(SamplerOption)
```

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 13.17 and can be used to compare the performance of the samplers.

13.4.14.3 Graph

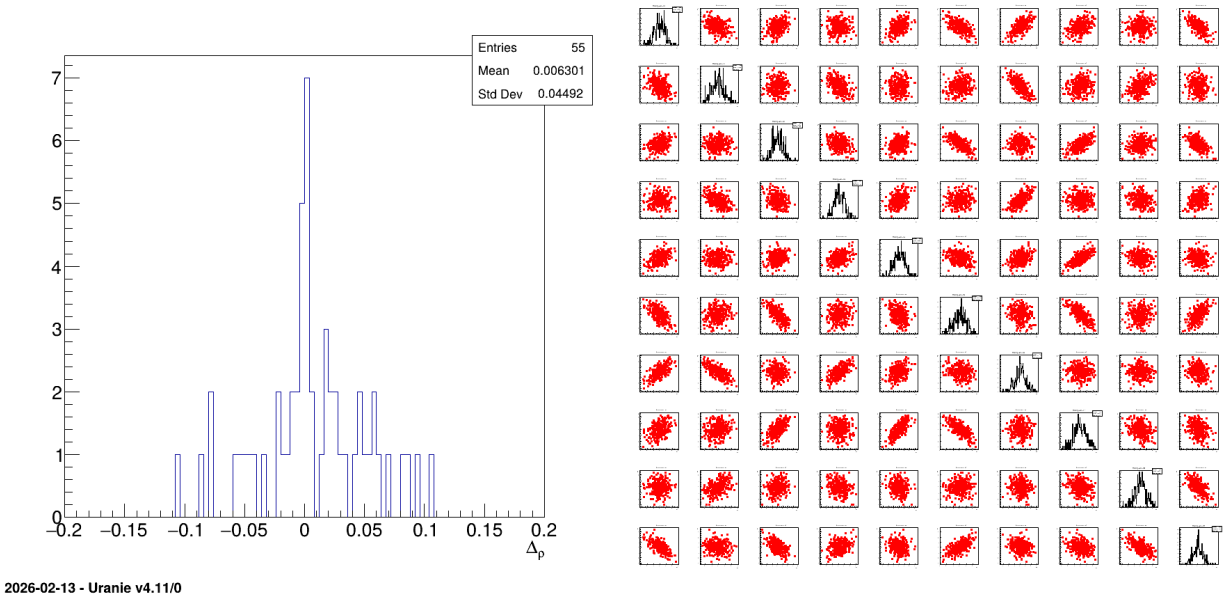


Figure 13.17: Graph of the macro **"samplingSingularCorrelationCase.py"**

13.5 Macros Launcher

13.5.1 Input/Output with vector and string: introduction to macros with multitype

13.5.1.1 Producing outputs

13.5.1.2 Reading inputs

13.5.2 Macro "launchCodeReadMultiTypeRow.py"

13.5.2.1 Objective

13.6 Macros Sensitivity

13.6.1 Macro "sensitivityBrutForceMethodFlowrate.py"

13.6.1.1 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 *Presentation of the problem*. 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.

13.6.1.2 Macro Uranie

```
"""
Example of Sobol estimation with a brut-force approach
"""
from URANIE import DataServer, Launcher, Sampler
import ROOT

def draw_bar_with_tuple(val, name, stitle):
    """Draw the results estimated with brut-force approach."""
    h_div = ROOT.TH1F("hDivdrawBarWithTuple", stitle, 3, 0, 3)
    h_div.SetCanExtend(ROOT.TH1.kXaxis)  # .SetBit(ROOT.TH1.kCanRebin)
    h_div.SetStats(0)

    if h_div != 0:
        h_div.SetBarWidth(0.45)
        h_div.SetBarOffset(0.1)
        h_div.SetMarkerColor(2)
        h_div.SetMarkerSize(2)
        h_div.SetFillColor(49)
        h_div.SetTitle(stitle)
        for ite, value in enumerate(val):
            h_div.Fill(name[ite], value)
        h_div.LabelsDeflate()
        h_div.LabelsOption(">u")
        h_div.SetMinimum(0.0)
        h_div.SetMaximum(1.0)
        ROOT.gStyle.SetPaintTextFormat("5.2f")
        h_div.Draw("bar2, text45")
    return h_div

nCond = 50
nbins = 10
# Create a DataServer.TDataServer
tds = DataServer.TDataServer()

print(" ******************************************************")
print(" ** sensitivityBrutForceMethodFlowrate nbins[%i] nCond[%i]" %
      (nbins, nCond))
print(" **")

# Create a DataServer.TDataServer
tds = DataServer.TDataServer()
# Add the eight attributes of the study with uniform law
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))
```

(continues on next page)

(continued from previous page)

```
nvar = tds.getNAttributes()
print(" ** nX["+str(nvar)+"]")
nS = nbins*nvar*nCond
print(" ** nS["+str(nS)+"]")


# sam = Sampler.TSampling(tds, "lhs", nS)
sam = Sampler.TQMC(tds, "halton", nS)
sam.generateSample()

# Load the function
ROOT.gROOT.LoadMacro("UserFunctions.C")

# Create a TLauncherFunction from a TDataServer and an analytical function
# Rename the outpout attribute "ymod"
tlf = Launcher.TLauncherFunction(tds, "flowrateModel",
                                 "rw:r:tu:tl:hu:hl:l:kw", "ymod")
# Evaluate the function on all the design of experiments
tlf.setDrawProgressBar(False)
tlf.run()

Canvas = ROOT.TCanvas("c1", "Graph for the Macro modeler", 5, 64, 1270, 667)
pad = ROOT.TPad("pad", "pad", 0, 0.03, 1, 1)
pad.Draw()
pad.Divide(2, 2)
pad.cd(1)
tds.computeStatistic("ymod")
tds.draw("ymod")
dstdy = tds.getAttribute("ymod").getStd()
svary = dstdy * dstdy

print(" ** ymod: std["+str(round(dstdy, 4))+"] vary["+str(round(svary, 1))+"]")

tds.getAttribute("ymod").setOutput()


ROOT.gStyle.SetOptStat(1)

valSobolCrt = []
sName = []

c = ROOT.TCanvas()
c.Divide(2)
c.cd(1)
for ivar in range(nvar):

    print(" ****************************")
    print(" *** "+str(tds.getAttribute(ivar).GetName()))

    svar = tds.getAttribute(ivar).GetName()

    if ivar == 0:
```

(continues on next page)

(continued from previous page)

```
        pad.cd(2)
    else:
        c.cd(1)

    tds.drawProfile("ymod:"+svar, "", "nclass="+str(nbins))
    hprofs = ROOT.gPad.GetPrimitive("Profile ymod:%s (Bin = %i )" %
                                    (svar, nbins+2))

    ntd = ROOT.TNtupleD("dd", "sjsjs", "i:x:m")
    ntd.SetMarkerColor(ROOT.kBlue)
    ntd.SetMarkerStyle(8)
    nnbins = hprofs.GetNbinsX()
    for i in range(1, nnbins+1):
        ntd.Fill(i-1, hprofs.GetBinCenter(i), hprofs.GetBinContent(i))

    tds.draw("ymod:"+svar)
    ntd.Draw("m:x", "", "same")

    if ivar == 0:
        pad.cd(3)
    else:
        c.cd(2)

    ntd.Draw("m")
    htemp = ROOT.gPad.GetPrimitive("htemp")

    dvarcond = htemp.GetRMS()

    # Tempory ROOT.TTree for histogram
    valSobolCrt.append(dvarcond*dvarcond / svary)
    sName.append(svar)

    print(" *** S1[ %s] Cond. Var.[%4.6g] -- [%1.6g]" %
          (svar, dvarcond*dvarcond, valSobolCrt[-1]))

    c.Modified()
    c.Update()
    c.SaveAs("SAFlowRateVersus"+svar+".png")

pad.cd(4)
hDiv = draw_bar_with_tuple(valSobolCrt, sName,
                           "Sensitivity Indexes: ymod  [Brute-Force Method]")
```

Each parameter is related to the `TDataServer` as a `TAttribute` and obeys an uniform law on specific interval:

```
tds = DataServer.TDataServer()
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
```

(continues on next page)

(continued from previous page)

```
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))
```

A design-of-experiments is built with a "Halton" method (n_S = 4000):

```
sam = Sampler.TQMC(tds, "halton", nS)
sam.generateSample()
```

The function `flowrateModel` is loaded from the macro `UserFunctions.C` (the file can be found in `${URANIESYS}/share/uranie/macros`):

```
ROOT.gROOT.LoadMacro("UserFunctions.C")
```

The `flowrateModel` model is applied on previous variables:

```
tlf = Launcher.TLauncherFunction(tds, "flowrateModel",
                                 "rw:r:tu:tl:hu:hl:l:kw", "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)
...
nnbins = hprofs.GetNbinsX()
for i in range(1,nnbins+1): 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")
htemp = ROOT.gPad.GetPrimitive("htemp")

dvarcond = htemp.GetRMS()
valSobolCrt.append(dvarcond*dvarcond / svary)
```

13.6.1.3 Graph

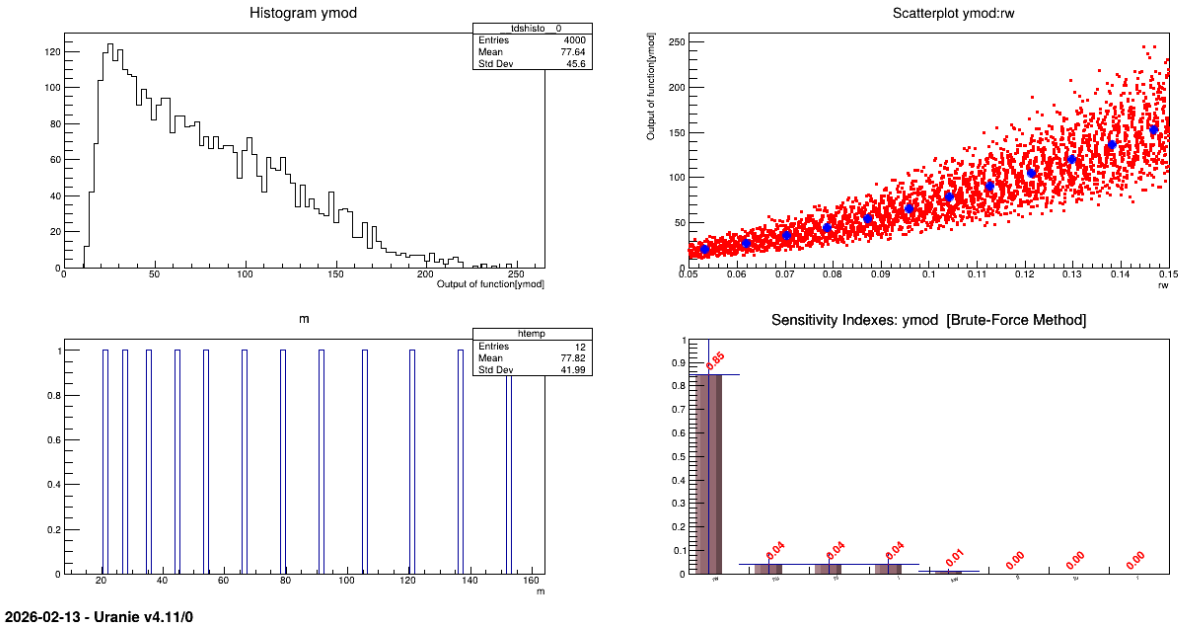


Figure 13.18: Graph of the macro **"sensitivityBrutForceMethodFlowrate.py"**

13.6.1.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

Info in <TCanvas::Print>: png file SAFlowRateVersusrw.png has been created
Info in <TCanvas::Print>: png file SAFlowRateVersusr.png has been created
Info in <TCanvas::Print>: png file SAFlowRateVersustu.png has been created
Info in <TCanvas::Print>: png file SAFlowRateVersustl.png has been created
Info in <TCanvas::Print>: png file SAFlowRateVersushu.png has been created
Info in <TCanvas::Print>: png file SAFlowRateVersushl.png has been created
Info in <TCanvas::Print>: png file SAFlowRateVersusl.png has been created
Info in <TCanvas::Print>: png file SAFlowRateVersuskw.png has been created
 ****************************************************
 ** 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]
 ****************************
 *** r
 *** S1[ r] Cond. Var.[0.0624988] -- [3.00489e-05]
```

(continues on next page)

(continued from previous page)

```
****************************
*** tu
*** S1[ tu] Cond. Var.[0.0886501] -- [4.26223e-05]
****************************
*** tl
*** S1[ tl] Cond. Var.[0.0909604] -- [4.37331e-05]
****************************
*** hu
*** S1[ hu] Cond. Var.[88.368] -- [0.0424867]
****************************
*** hl
*** S1[ hl] Cond. Var.[88.2039] -- [0.0424078]
****************************
*** l
*** S1[ l] Cond. Var.[84.9543] -- [0.0408454]
****************************
*** kw
*** S1[ kw] Cond. Var.[20.8848] -- [0.0100413]
```

13.6.2 Macro “sensitivityFiniteDifferencesFunctionFlowrate.py”

13.6.2.1 Objective

The objective of this macro is to compute the finite differences indexes on a function.

13.6.2.2 Macro Uranie

```
"""
Example of finite difference approach to the flowrate model
"""
from URANIE import DataServer, Sensitivity
import ROOT

# loading the flowrateModel function
ROOT.gROOT.LoadMacro("UserFunctions.C")

# Define the DataServer  and add the attributes (stochastic variables here)
tds = DataServer.TDataServer("tdsflowrate", "DataBase flowrate")
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.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)
```

(continues on next page)

(continued from previous page)

```
tds.getAttribute("hl").setDefaultValue(760.0)
tds.getAttribute("l").setDefaultValue(1400.0)
tds.getAttribute("kw").setDefaultValue(10500.0)

# Create a TFiniteDifferences object
tfindef = Sensitivity.TFiniteDifferences(tds, "flowrateModel",
                                         "rw:r:tu:tl:hu:hl:l:kw",
                                         "flowrateModel", "steps=1%")
tfindef.setDrawProgressBar(False)
tfindef.computeIndexes()
matRes = tfindef.getSensitivityMatrix()
matRes.Print()
```

The function `flowrateModel` is loaded from the macro `UserFunctions.C` (the file can be found in `${URANIESYS}/share/uranie/macros`)

```
ROOT.gROOT.LoadMacro("UserFunctions.C")
```

Each parameter is related to the `TDataServer` as a `TAttribute` and obeys an uniform law on specific interval:

```
tds = DataServer.TDataServer("tdsflowrate", "DataBase flowrate")
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.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:

```
tfindef = Sensitivity.TFiniteDifferences(tds, "flowrateModel",
                                         "rw:r:tu:tl:hu:hl:l:kw",
                                         "flowrateModel", "steps=1%")
```

Computation of sensitivity indexes:

```
tfindef.computeIndexes()
```

13.6.2.3 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026


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
```

13.6.3 Macro “sensitivityDataBaseFlowrate.py”

13.6.3.1 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
```

(continues on next page)

(continued from previous page)

```
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
```

13.6.3.2 Macro Uranie

```
"""
Example of sensitivity analysis through linear regression
"""
from URANIE import DataServer, Sensitivity
import ROOT

# Create a DataServer.TDataServer
tds = DataServer.TDataServer()
# Load a database in an ASCII file
tds.fileDataRead("flowrateUniformDesign.dat")

# Graph
Canvas = ROOT.TCanvas("c2", "Graph for the Macro", 5, 64, 1270, 667)
# Visualisation
tds.Draw("ystar:rw")

# Sensitivity analysis
treg = Sensitivity.TRegression(tds, "rw:r:tu:tl:hu:hl:l:kw", "ystar", "src")
treg.computeIndexes()

treg.drawIndexes("Flowrate", "", "hist, first")
# treg.getResultTuple().Scan()

# Graph
c = ROOT.gROOT.FindObject("__sensitivitycan__0")
can = ROOT.TCanvas("c1", "Graph for the Macro sensitivityDataBaseFlowrate",
                   5, 64, 1270, 667)
pad = ROOT.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:

```
treg = Sensitivity.TRegression(tds, "rw:r:tu:tl:hu:hl:l:kw", "ystar", "src")
treg.computeIndexes()
```

13.6.3.3 Graph

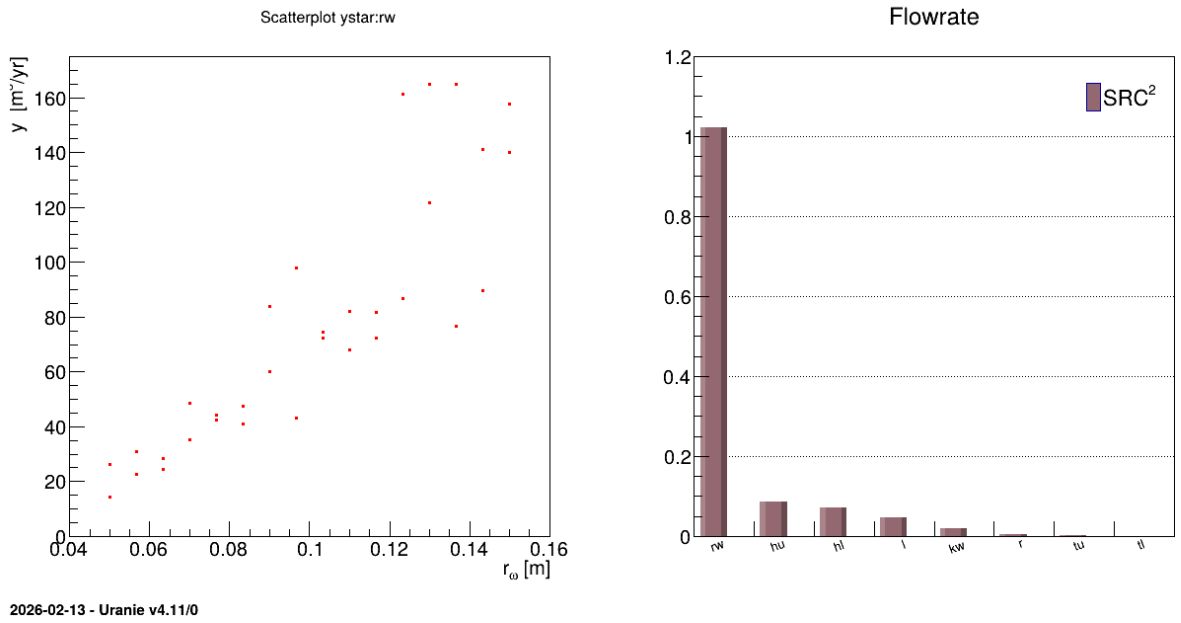


Figure 13.19: Graph of the macro **"sensitivityDataBaseFlowrate.py"**

13.6.4 Macro "sensitivityFASTFunctionFlowrate.py"

13.6.4.1 Objective

The objective of this macro is to perform a Fast sensitivity analysis on a set of eight parameters used in the `flowrate-Model` model described in *Presentation of the problem*.

13.6.4.2 Macro Uranie

```
"""
Example of the flowrate function sensitivity analysis
"""
from URANIE import DataServer, Sensitivity
import ROOT

ROOT.gROOT.LoadMacro("UserFunctions.C")

# Define the DataServer
tds = DataServer.TDataServer("tdsflowreate", "DataBase flowreate")
```

(continues on next page)

(continued from previous page)

```
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))

# \param Size of a sampling.
nS = 4000
# Graph
fast = Sensitivity.TFast(tds, "flowrateModel", nS)
fast.setDrawProgressBar(False)
fast.computeIndexes("graph")

fast.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}/share/uranie/macros`)

```
ROOT.gROOT.LoadMacro("UserFunctions.C")
```

Each parameter is related to the `TDataServer` as a `TAttribute` and obeys an uniform law on specific interval:

```
tds = DataServer.TDataServer("tdsflowreate", "DataBase flowreate")
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.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):

```
fast = Sensitivity.TFast(tds, "flowrateModel", nS)
```

Computation of sensitivity indexes:

```
fast.computeIndexes("graph")
```

13.6.4.3 Graph

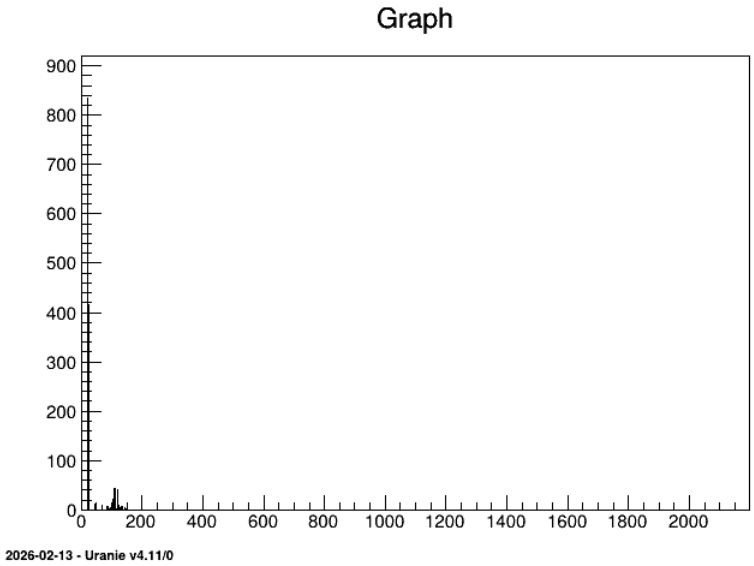


Figure 13.20: Graph of the macro **"sensitivityFASTFunctionFlowrate.py"**

13.6.4.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8531]
 <URANIE::WARNING> TDataServer::getTuple Error : There is no tree!
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8531]
 <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 *
*****************************************************************
```

(continues on next page)

(continued from previous page)

```
*        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
```

13.6.5 Macro “sensitivityRBDFunctionFlowrate.py”

13.6.5.1 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 *Presentation of the problem*.

13.6.5.2 Macro Uranie

```
"""
Example of RDB analysis on the flowrate function
"""
from URANIE import DataServer, Sensitivity
import ROOT

ROOT.gROOT.LoadMacro("UserFunctions.C")

# Define the DataServer
tds = DataServer.TDataServer("tdsflowrate", "DataBase flowrate")
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))

# Size of a sampling.
nS = 4000
# Graph
trbd = Sensitivity.TRBD(tds, "flowrateModel", nS)
trbd.setDrawProgressBar(False)
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}/share/uranie/macros`)

```
ROOT.gROOT.LoadMacro("UserFunctions.C")
```

Each parameter is related to the `TDataServer` as a `TAttribute` and obeys an uniform law on specific interval

```
tds = DataServer.TDataServer("tdsflowrate", "DataBase flowrate")
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.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 = Sensitivity.TRBD(tds, "flowrateModel", nS)
```

Computation of sensitivity indexes:

```
trbd.computeIndexes("graph")
```

13.6.5.3 Graph

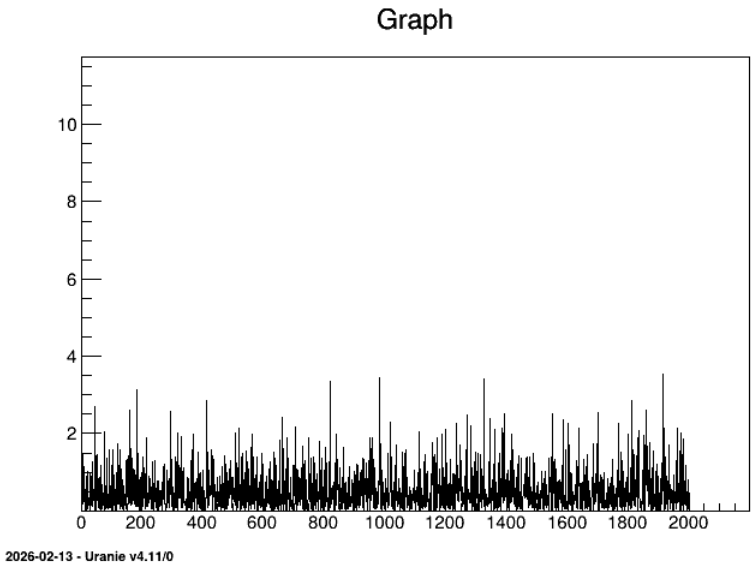


Figure 13.21: Graph of the macro **"sensitivityRBDFunctionFlowrate.py"**

13.6.5.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
```

(continues on next page)

(continued from previous page)

```
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8531]
 <URANIE::WARNING> TDataServer::getTuple Error : There is no tree!
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8531]
 <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
```

13.6.6 Macro “sensitivityMorrisFunctionFlowrate.py”

13.6.6.1 Objective

The objective of this macro is to perform a Morris sensitivity analysis on a set of eight parameters used in the `flowrate-Model` model described in *Presentation of the problem*.

13.6.6.2 Macro Uranie

```
"""
Example of Morris estimation on flowrate
"""
from URANIE import DataServer, Sensitivity
import ROOT

ROOT.gROOT.LoadMacro("UserFunctions.C")

# Define the DataServer
tds = DataServer.TDataServer("tdsflowreate", "DataBase flowreate")
```

(continues on next page)

(continued from previous page)

```
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))

nreplique = 3
nlevel = 10
scmo = Sensitivity.TMorris(tds, "flowrateModel", nreplique, nlevel)
scmo.setDrawProgressBar(False)
scmo.generateSample()

tds.exportData("_morris_sampling_.dat")
scmo.computeIndexes()

tds.exportData("_morris_launching_.dat")

ntresu = scmo.getMorrisResults()
ntresu.Scan("*")

# Graph
canmoralltraj = ROOT.gROOT.FindObject("canmoralltraj")
can = ROOT.TCanvas("c1", "Graph of sensitivityMorrisFunctionFlowrate",
                   5, 64, 1270, 667)
pad = ROOT.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/uranie/macros`)

```
ROOT.gROOT.LoadMacro("UserFunctions.C")
```

Each parameter is related to the `TDataServer` as a `TAttribute` and obeys an uniform law on specific interval:

```
tds = DataServer.TDataServer("tdsflowreate", "DataBase flowreate")
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.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)

```
scmo = Sensitivity.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()
```

13.6.6.3 Graph

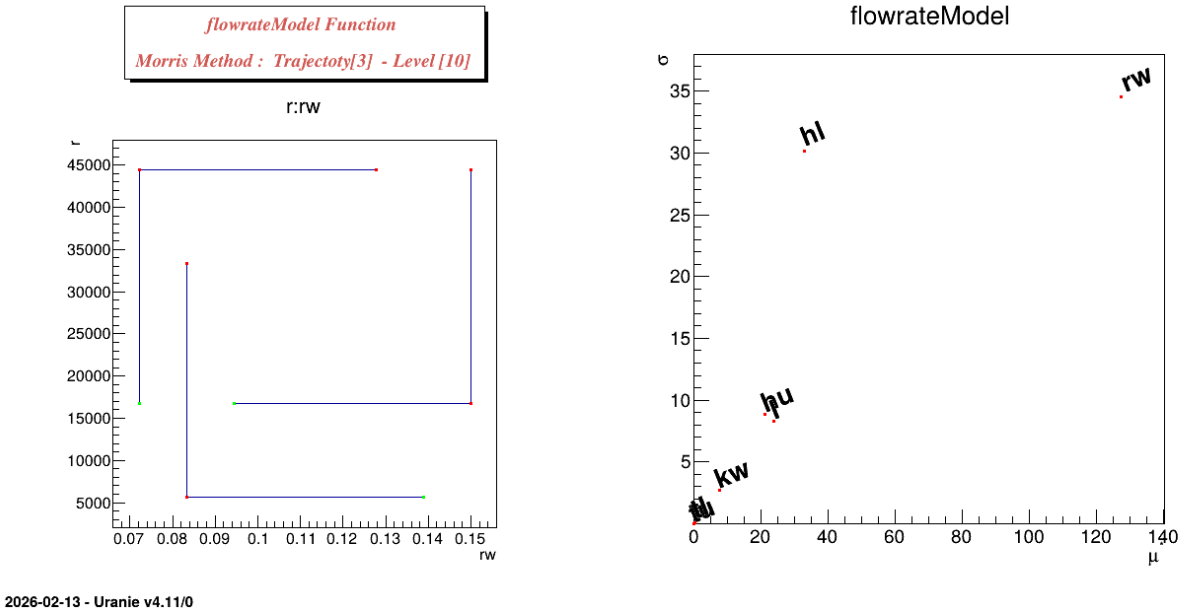


Figure 13.22: Graph of the macro **"sensitivityMorrisFunctionFlowrate.py"**

13.6.6.4 Console

```
*************************************************************************
*    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 *
```

(continues on next page)

(continued from previous page)

```
*        6 *         l * flowrateM * -23.89328 * 23.893280 * 8.2781934 *
*        7 *        kw * flowrateM * 7.5766167 * 7.5766167 * 2.7457665 *
*****************************************************************************
```

13.6.7 Macro "sensitivityMorrisFunctionFlowrateRunner.py"

13.6.7.1 Objective

The objective of this macro is to perform a Morris sensitivity analysis on a set of eight parameters used in the `flowrate-Model` model described in *Presentation of the problem*, but this time using the Relauncher architecture.

13.6.7.2 Macro Uranie

```
"""
Example of Morris analysis on flowrate with a Relauncher approach
"""
from URANIE import DataServer, Relauncher, Sensitivity
import ROOT

ROOT.gROOT.LoadMacro("UserFunctions.C")

# Define the DataServer
rw = DataServer.TUniformDistribution("rw", 0.05, 0.15)
r = DataServer.TUniformDistribution("r", 100.0, 50000.0)
tu = DataServer.TUniformDistribution("tu", 63070.0, 115600.0)
tl = DataServer.TUniformDistribution("tl", 63.1, 116.0)
hu = DataServer.TUniformDistribution("hu", 990.0, 1110.0)
hl = DataServer.TUniformDistribution("hl", 700.0, 820.0)
lvar = DataServer.TUniformDistribution("l", 1120.0, 1680.0)
kw = DataServer.TUniformDistribution("kw", 9855.0, 12045.0)

# Create the evaluator
code = Relauncher.TCIntEval("flowrateModel")
# Create output attribute
yout = DataServer.TAttribute("flowrateModel")
# Provide input/output attributes to the assessor
code.addInput(rw)
code.addInput(r)
code.addInput(tu)
code.addInput(tl)
code.addInput(hu)
code.addInput(hl)
code.addInput(lvar)
code.addInput(kw)
code.addOutput(yout)

run = Relauncher.TSequentialRun(code)  # Replace to distribute computation
run.startSlave()
if run.onMaster():
    # Create the dataserver
    tds = DataServer.TDataServer("sobol", "foo bar pouet chocolat")
    tds.addAttribute(rw)
```

(continues on next page)

(continued from previous page)

```
    tds.addAttribute(r)
    tds.addAttribute(tu)
    tds.addAttribute(tl)
    tds.addAttribute(hu)
    tds.addAttribute(hl)
    tds.addAttribute(lvar)
    tds.addAttribute(kw)

    # Create the Morris object
    nreplique = 3
    nlevel = 10
    scmo = Sensitivity.TMorris(tds, run, nreplique, nlevel)
    scmo.setDrawProgressBar(False)
    scmo.generateSample()

    tds.exportData("_morris_sampling_.dat")
    scmo.computeIndexes()

    tds.exportData("_morris_launching_.dat")

    ntresu = scmo.getMorrisResults()
    ntresu.Scan("*")

    # Graph
    canmoralltraj = ROOT.gROOT.FindObject("canmoralltraj")
    can = ROOT.TCanvas("c1", "Graph sensitivityMorrisFunctionFlowrateRunner",
                       5, 64, 1270, 667)
    pad = ROOT.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/uranie/macros`)

```
ROOT.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
rw = DataServer.TUniformDistribution("rw", 0.05, 0.15)
r = DataServer.TUniformDistribution("r", 100.0, 50000.0)
tu = DataServer.TUniformDistribution("tu", 63070.0, 115600.0)
tl = DataServer.TUniformDistribution("tl", 63.1, 116.0)
hu = DataServer.TUniformDistribution("hu", 990.0, 1110.0)
hl = DataServer.TUniformDistribution("hl", 700.0, 820.0)
lvar = DataServer.TUniformDistribution("l", 1120.0, 1680.0)
kw = DataServer.TUniformDistribution("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. On the contrary to the C++ script, python cannot use methods such as `setInputs` or `setOutputs` and the inputs and outputs have to be included ony-by-one.

```
# Create the evaluator
code = Relauncher.TCIntEval("flowrateModel")
# Create output attribute
yout = DataServer.TAttribute("flowrateModel")
# Provide input/output attributes to the assessor
code.addInput(rw)
code.addInput(r)
code.addInput(tu)
code.addInput(tl)
code.addInput(hu)
code.addInput(hl)
code.addInput(lvar)
code.addInput(kw)
code.addOutput(yout)

run = Relauncher.TSequentialRun(code)   # Replace to distribute 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)

```
scmo = Sensitivity.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.

13.6.7.3 Graph

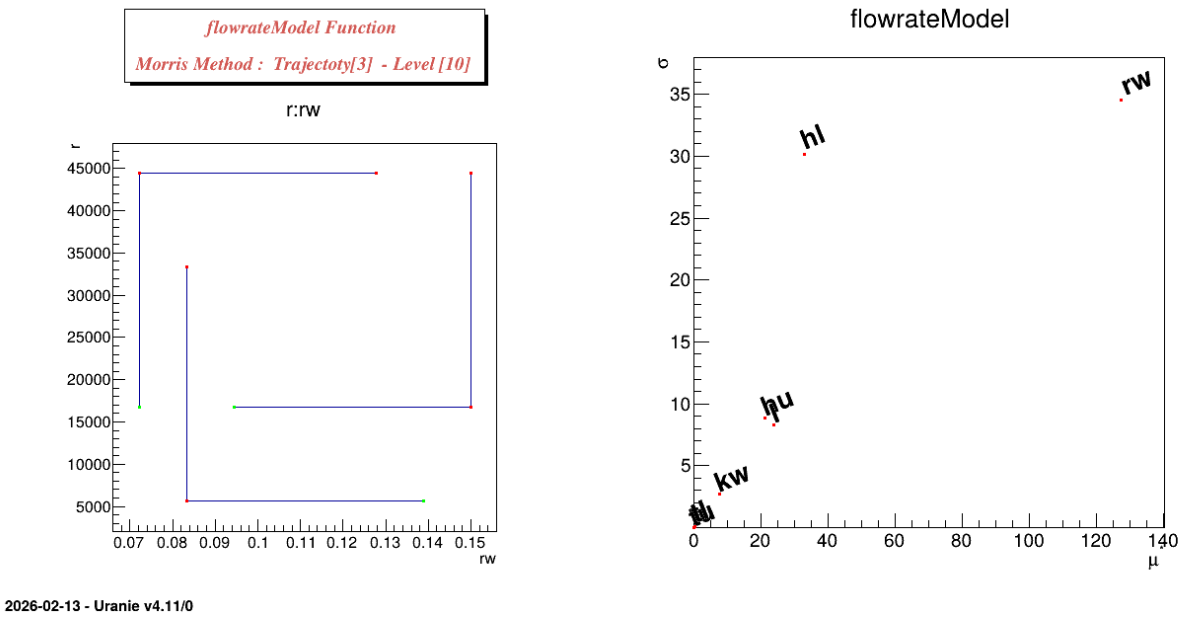


Figure 13.23: Graph of the macro **"sensitivityMorrisFunctionFlowrateRunner.py"**

13.6.7.4 Console

```
*************************************************************************
*    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 *
*************************************************************************
```

13.6.8 Macro "sensitivityRegressionFunctionFlowrate.py"

13.6.8.1 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 *Presentation of the problem* and "loaded" through the macro `UserFunctions.C` (the file can be found in `${URANIESYS}/share/uranie/macros`). Function `flowrateModel` uses the eight variables defined in *Presentation of the problem* and set in the main macro.

13.6.8.2 Macro Uranie

```
"""
Example of regression approach on the flowrate function
"""
from URANIE import DataServer, Launcher, Sampler, Sensitivity
import ROOT

ROOT.gROOT.LoadMacro("UserFunctions.C")

# Define the DataServer
tds = DataServer.TDataServer("tdsflowreate", "DataBase flowreate")
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))

# Size of a sampling.
nS = 4000

sampling = Sampler.TSampling(tds, "lhs", nS)
sampling.generateSample()

tlf = Launcher.TLauncherFunction(tds, "flowrateModel")
tlf.setDrawProgressBar(False)
tlf.run()

treg = Sensitivity.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\"","colsize=5 col=6:8::9:8:8:8")

can = ROOT.TCanvas("c1", "Graph 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(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))
```

The sampling is generated on 4000 patterns with a LHS method:

```
sampling = Sampler.TSampling(tds, "lhs", nS)
sampling.generateSample()
```

Function `flowrateModel` is set to perform calculation on the sampling:

```
tlf = Launcher.TLauncherFunction(tds, "flowrateModel")
tlf.run()
```

The regression is performed over all variables:

```
treg = Sensitivity.TRegression(tds, "rw:r:tu:tl:hu:hl:l:kw",
                               "flowrateModel", "SRC")
treg.computeIndexes()
```

Sensitivity indexes are then displayed through an histogram:

```
can = ROOT.TCanvas("c1", "Graph sensitivityRegressionFunctionFlowrate",
                   5, 64, 1270, 667)
treg.drawIndexes("Flowrate", "", "hist, first, nonewcanv")
```

13.6.8.3 Graph

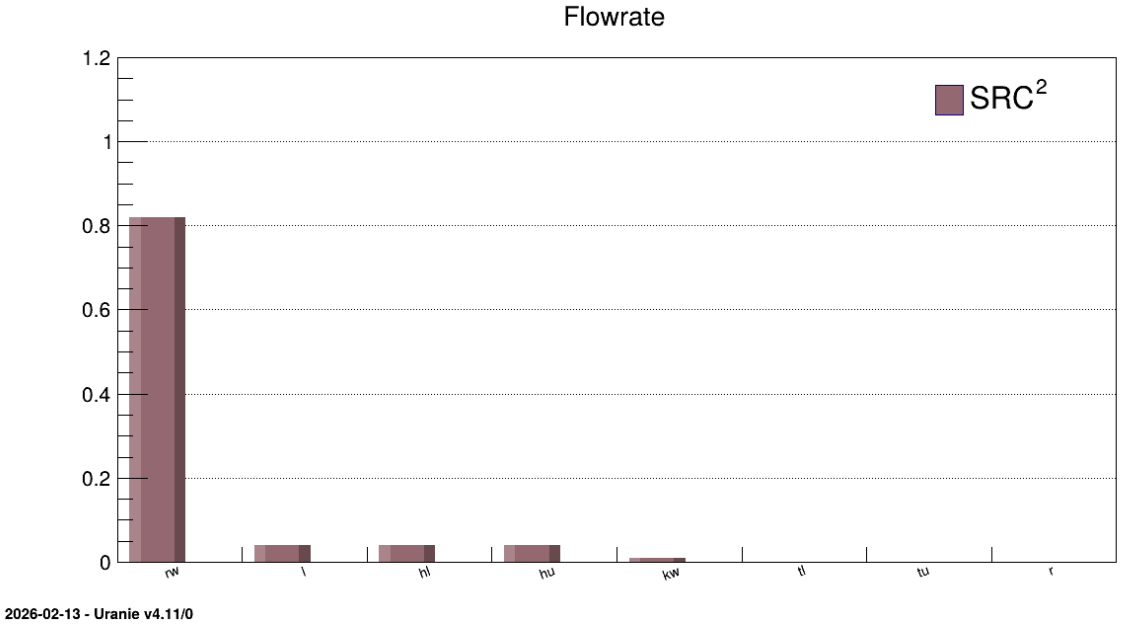


Figure 13.24: Graph of the macro **"sensitivityRegressionFunctionFlowrate.py"**

13.6.8.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
```

(continues on next page)

(continued from previous page)

```
                    Date: Thu Feb 12, 2026

*******************************************************************************
*    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
```

13.6.9 Macro “sensitivitySobolFunctionFlowrate.py”

13.6.9.1 Objective

The objective of this macro is to perform Sobol sensitivity analysis on a set of eight parameters used in the `flowrate-Model` model described in *Presentation of the problem*.

13.6.9.2 Macro Uranie

```
"""
Example of Sobol estimation for the flowrate function
"""
from URANIE import DataServer, Sensitivity
import ROOT

ROOT.gROOT.LoadMacro("UserFunctions.C")

# Define the DataServer
tds = DataServer.TDataServer("tdsflowreate", "DataBase flowreate")
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
```

(continues on next page)

(continued from previous page)

```
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))

ns = 100000
tsobol = Sensitivity.TSobol(tds, "flowrateModel", ns, "rw:r:tu:tl:hu:hl:l:kw",
                            "flowrateModel", "DummyPython")
tsobol.setDrawProgressBar(False)
tsobol.computeIndexes()

tsobol.getResultTuple().Scan("*", "Algo==\"--first--\" || Algo==\"--total--\"")

cc = ROOT.TCanvas("c1", "histgramme", 5, 64, 1270, 667)
pad = ROOT.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")

ROOT.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}/share/uranie/macros`)

```
ROOT.gROOT.LoadMacro("UserFunctions.C")
```

Each parameter is related to the `TDataServer` as a `TAttribute` and obeys an uniform law on specific interval:

```
tds = DataServer.TDataServer("tdsflowreate", "DataBase flowreate")
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.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 = Sensitivity.TSobol(tds, "flowrateModel", ns, "rw:r:tu:tl:hu:hl:l:kw",
                            "flowrateModel", "DummyPython")
```

The last argument is the option field, which in most cases is empty. Here it is filled with "DummyPython" which helps specify to python which constructor to choose. There are indeed several possible constructors these 5 five first arguments, but C++ can make the difference between them as the literal members are either `std::string`, `ROOT::TString`, `Char_t*` or even `Option_t*`. For python, these format are all `PyString`, so the sixth argument is compulsory to

disentangle the possibilities.

Computation of the sensitivity indexes:

```
tsobol.computeIndexes()
```

Data are exported from the `TDataServer` to an ASCII file:

```
tds.exportData("_onlyMandN_sobol_launching_.dat",
               "rw:r:tu:tl:hu:hl:l:kw:flowrateModel",
               "sobol__n__iter__tdsflowreate < 100")
```

13.6.9.3 Graph

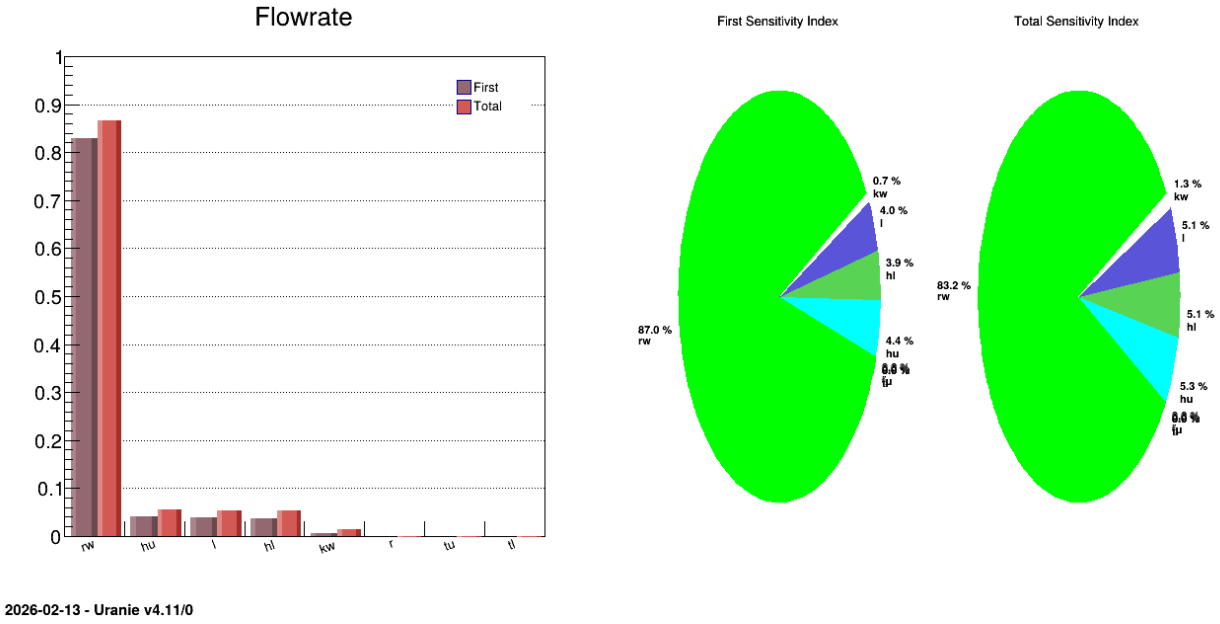


Figure 13.25: Graph of the macro **"sensitivitySobolFunctionFlowrate.py"**

13.6.9.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8531]
 <URANIE::WARNING> TDataServer::getTuple Error : There is no tree!
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${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 *
```

(continues on next page)

(continued from previous page)

```
*       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
```

13.6.10 Macro "sensitivitySobolFunctionFlowrateRunner.py"

13.6.10.1 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 *Presentation of the problem*, but this time using the Relauncher architecture.

13.6.10.2 Macro Uranie

```
"""
Example of Sobol estimation for the flowrate function with Relauncher approach
"""
from URANIE import DataServer, Relauncher, Sensitivity
import ROOT

ROOT.gROOT.LoadMacro("UserFunctions.C")

# Define the DataServer
rw = DataServer.TUniformDistribution("rw", 0.05, 0.15)
r = DataServer.TUniformDistribution("r", 100.0, 50000.0)
tu = DataServer.TUniformDistribution("tu", 63070.0, 115600.0)
tl = DataServer.TUniformDistribution("tl", 63.1, 116.0)
hu = DataServer.TUniformDistribution("hu", 990.0, 1110.0)
hl = DataServer.TUniformDistribution("hl", 700.0, 820.0)
lvar = DataServer.TUniformDistribution("l", 1120.0, 1680.0)
kw = DataServer.TUniformDistribution("kw", 9855.0, 12045.0)

# Create the evaluator
code = Relauncher.TCIntEval("flowrateModel")
# Create output attribute
yout = DataServer.TAttribute("flowrateModel")
# Provide input/output attributes to the assessor
code.addInput(rw)
code.addInput(r)
code.addInput(tu)
code.addInput(tl)
code.addInput(hu)
code.addInput(hl)
code.addInput(lvar)
code.addInput(kw)
code.addOutput(yout)

run = Relauncher.TSequentialRun(code)  # Replace to distribute computation
run.startSlave()
if run.onMaster():
    # Create the dataserver
    tds = DataServer.TDataServer("sobol", "foo bar pouet chocolat")
    code.addAllInputs(tds)

    # Create the sobol object
    ns = 100000
    tsobol = Sensitivity.TSobol(tds, run, ns)
    tsobol.setDrawProgressBar(ROOT.kFALSE)
    tsobol.computeIndexes()
```

(continues on next page)

(continued from previous page)

```
    cc = ROOT.TCanvas("c1", "histgramme", 5, 64, 1270, 667)
    pad = ROOT.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")

    tds.exportData("_sobol_launching_.dat")
```

The function `flowrateModel` is loaded from the macro `UserFunctions.C` (the file can be found in `${URANIESYS}/share/uranie/macros`)

```
ROOT.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
rw = DataServer.TUniformDistribution("rw", 0.05, 0.15)
r = DataServer.TUniformDistribution("r", 100.0, 50000.0)
tu = DataServer.TUniformDistribution("tu", 63070.0, 115600.0)
tl = DataServer.TUniformDistribution("tl", 63.1, 116.0)
hu = DataServer.TUniformDistribution("hu", 990.0, 1110.0)
hl = DataServer.TUniformDistribution("hl", 700.0, 820.0)
lvar = DataServer.TUniformDistribution("l", 1120.0, 1680.0)
kw = DataServer.TUniformDistribution("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
code = Relauncher.TCIntEval("flowrateModel")
# Create output attribute
yout = DataServer.TAttribute("flowrateModel")
# Provide input/output attributes to the assessor
code.addInput(rw)
code.addInput(r)
code.addInput(tu)
code.addInput(tl)
code.addInput(hu)
code.addInput(hl)
code.addInput(lvar)
code.addInput(kw)
code.addOutput(yout)

run = Relauncher.TSequentialRun(code)  # Replace to distribute 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 = Sensitivity.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")
```

13.6.10.3 Graph

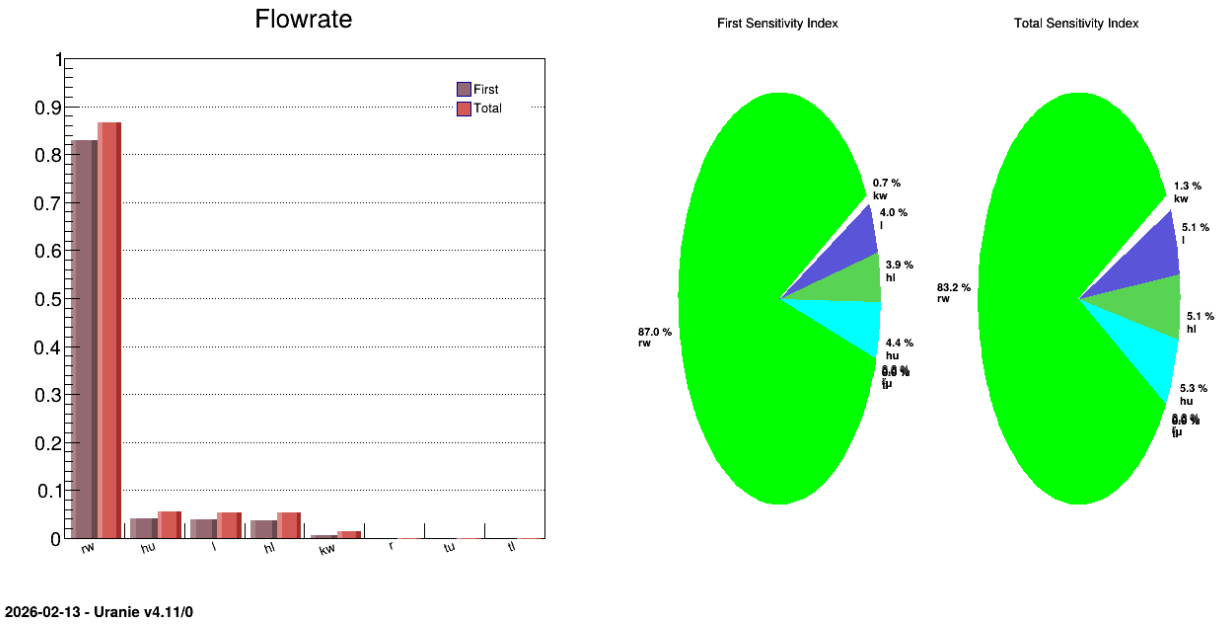


Figure 13.26: Graph of the macro **"sensitivitySobolFunctionFlowrateRunner.py"**

13.6.10.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8531]
 <URANIE::WARNING> TDataServer::getTuple Error : There is no tree!
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${SOURCEDIR}/meTIER/sampler/souRCE/TSamplerStochastic.cxx]␣
```

(continues on next page)

(continued from previous page)

```
↪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]
```

13.6.11 Macro “sensitivityRegressionLeveLE.py”

> **Warning**
>
> The `levele` command will be installed on your machine only if a Fortran compiler is found

13.6.11.1 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.

13.6.11.2 Macro Uranie

```
"""
Example of regression analysis on the levelE code
"""
import sys
from ctypes import c_double  # for ROOT version greater or equal to 6.20
import numpy as np
from URANIE import Launcher, Sampler, Sensitivity
from URANIE import DataServer as DS
import ROOT

# Create DataServer and add input attributes
tds = DS.TDataServer("tds", "levelE usecase")
tds.addAttribute(DS.TUniformDistribution("t", 100, 1000))
tds.addAttribute(DS.TLogUniformDistribution("kl", 0.001, .01))
tds.addAttribute(DS.TLogUniformDistribution("kc", 1.0e-6, 1.0e-5))
tds.addAttribute(DS.TLogUniformDistribution("v1", 1.0e-3, 1.0e-1))
tds.addAttribute(DS.TUniformDistribution("l1", 100., 500.))
tds.addAttribute(DS.TUniformDistribution("r1", 1., 5.))
tds.addAttribute(DS.TUniformDistribution("rc1", 3., 30.))
tds.addAttribute(DS.TLogUniformDistribution("v2", 1.0e-2, 1.0e-1))
```

(continues on next page)

(continued from previous page)

```
tds.addAttribute(DS.TUniformDistribution("l2", 50., 200.))
tds.addAttribute(DS.TUniformDistribution("r2", 1., 5.))
tds.addAttribute(DS.TUniformDistribution("rc2", 3., 30.))
tds.addAttribute(DS.TLogUniformDistribution("w", 1.0e5, 1.0e7))

# Tell the code where to find attribute value in input file
sIn = "levelE_input_with_values_rows.in"
tds.getAttribute("t").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("kl").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("kc").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("v1").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("l1").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("r1").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("rc1").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("v2").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("l2").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("r2").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("rc2").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("w").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)

# Create DOE
ns = 1024
samp = Sampler.TSampling(tds, "lhs", ns)
samp.generateSample()

# How to read ouput files
out = Launcher.TOutputFileRow("_output_levelE_withRow_.dat")
# Tell the output file that attribute IS a vector and is SECOND column
out.addAttribute(DS.TAttribute("y", DS.TAttribute.kVector), 2)

# Creation of Launcher.TCode
myc = Launcher.TCode(tds, " levele 2> /dev/null")
myc.addOutputFile(out)

# Run the code
tl = Launcher.TLauncher(tds, myc)
tl.run(args.runOpt) # args.runOpt = "" or "nointermed"

# Launch Regression
tsen = Sensitivity.TRegression(tds, "t:kl:kc:v1:l1:r1:rc1:v2:l2:r2:rc2:w",
                               "y", "SRCSRRC")
tsen.computeIndexes()
res = tsen.getResultTuple()

# Plotting mess
tps = np.array([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],
               dtype=float)
colors = [1, 2, 3, 4, 6, 7, 8, 15, 30, 38, 41, 46]

c2 = ROOT.TCanvas("c2", "c2", 5, 64, 1600, 500)
```

(continues on next page)

(continued from previous page)

```
pad = ROOT.TPad("pad", "pad", 0, 0.03, 1, 1)
pad.Draw()
pad.Divide(3, 1)
pad.cd(1)
ROOT.gPad.SetLogx()
ROOT.gPad.SetGrid()
mg = ROOT.TMultiGraph()
res.Draw("Value", "Inp==\"__R2__\" && Order==\"Total\" && Method==\"SRRC^2\"",
         "goff")
data3 = res.GetV1()

gr3 = ROOT.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")
data4 = res.GetV1()

gr4 = ROOT.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
ROOT.gStyle.SetLegendBorderSize(0)
ROOT.gStyle.SetFillStyle(0)

lg = ROOT.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)
ROOT.gPad.SetLogx()
ROOT.gPad.SetGrid()
mg2 = ROOT.TMultiGraph()

names = ["t", "kl", "kc", "v1", "l1", "r1", "rc1",
         "v2", "l2", "r2", "rc2", "w"]
src = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
leg = ROOT.TLegend(0.25, 0.3, 0.45, 0.89, "Cumulative contributions")
leg.SetTextSize(0.035)
for igr in range(12):

```

(continues on next page)

(continued from previous page)

```
    sel = "Inp==\""+names[igr]+"\""
    sel += "&& Order==\"Total\" && Method==\"SRC^2\" && Algo!=\"--rho^2--\""
    res.Draw("Value", sel, "goff")
    data = res.GetV1()
    src[igr] = ROOT.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 ip in range(26):

        x = c_double(0)
        y = c_double(0)  # For ROOT lower than 6.20, user ROOT.Double
        if igr != 0:
            src[igr-1].GetPoint(ip+1, x, y)
        src[igr].SetPoint(ip+1, tps[ip], y.value+data[ip])

    src[igr].SetPoint(27, tps[25]*1.000000001, 0)
    leg.AddEntry(src[igr], names[igr], "f")

for igr2 in range(11, -1, -1):
    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)
ROOT.gPad.SetLogx()
ROOT.gPad.SetGrid()
mg3 = ROOT.TMultiGraph()
srrc = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

for igr in range(12):
    sel = "Inp==\""+names[igr]+"\""
    sel += "&& Order==\"Total\" && Method==\"SRRC^2\" && Algo!=\"--rho^2--\""
    res.Draw("Value", sel, "goff")
    data = res.GetV1()
    srrc[igr] = ROOT.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 ip in range(26):
        x = c_double(0)
        y = c_double(0)  # For ROOT lower than 6.20, user ROOT.Double
        if igr != 0:
            srrc[igr-1].GetPoint(ip+1, x, y)
```

(continues on next page)

(continued from previous page)

```
        srrc[igr].SetPoint(ip+1, tps[ip], y.value+data[ip])

    srrc[igr].SetPoint(27, tps[25]*1.000000001, 0)
    srrc[igr].SetTitle(names[igr])

for igr2 in range(11, -1, -1):
    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(DS.TAttribute("y", DS.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).

13.6.11.3 Graph

The results of the previous macro is shown in Figure 13.27, 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.

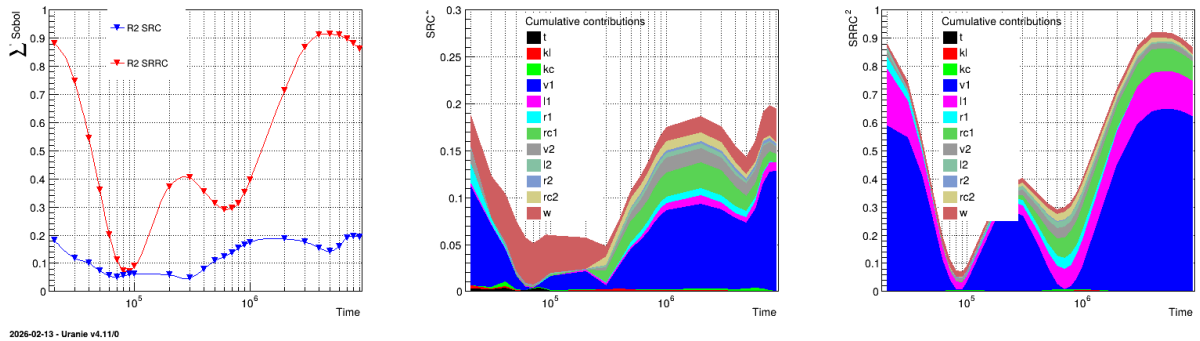


Figure 13.27: Graph of the macro **"sensitivityRegressionLeveLE.py"**

13.6.12 Macro “sensitivitySobolLeveLE.py”

> **Warning**
>
> The `levele` command will be installed on your machine only if a Fortran compiler is found

13.6.12.1 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.

13.6.12.2 Macro Uranie

```
"""
Example of Sobol estimation on the levelE code
"""
import sys
import numpy as np
from URANIE import Launcher, Sensitivity
from URANIE import DataServer as DS
import ROOT

# Define the DataServer
tds = DS.TDataServer("tdsLevelE", "levele")
tds.addAttribute(DS.TUniformDistribution("t", 100, 1000))
tds.addAttribute(DS.TLogUniformDistribution("kl", 0.001, 0.01))
tds.addAttribute(DS.TLogUniformDistribution("kc", 0.000001, 0.00001))
tds.addAttribute(DS.TLogUniformDistribution("v1", 0.001, 0.1))
tds.addAttribute(DS.TUniformDistribution("l1", 100, 500))
tds.addAttribute(DS.TUniformDistribution("r1", 1, 5))
tds.addAttribute(DS.TUniformDistribution("rc1", 3, 30))
tds.addAttribute(DS.TLogUniformDistribution("v2", 0.01, 0.1))
tds.addAttribute(DS.TUniformDistribution("l2", 50, 200))
tds.addAttribute(DS.TUniformDistribution("r2", 1, 5))
tds.addAttribute(DS.TUniformDistribution("rc2", 3, 30))
tds.addAttribute(DS.TLogUniformDistribution("w", 100000, 10000000))

# Tell the code where to find attribute value in input file
sIn = "levelE_input_with_values_rows.in"
tds.getAttribute("t").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("kl").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("kc").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("v1").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("l1").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("r1").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("rc1").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("v2").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("l2").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("r2").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
tds.getAttribute("rc2").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)
```

(continues on next page)

(continued from previous page)

```
tds.getAttribute("w").setFileKey(sIn, "", "%e", DS.TAttributeFileKey.kNewRow)

# How to read ouput files
out = Launcher.TOutputFileRow("_output_levelE_withRow_.dat")
# Tell the output file that attribute IS a vector and is SECOND column
out.addAttribute(DS.TAttribute("y", DS.TAttribute.kVector), 2)

# Creation of Launcher.TCode
myc = Launcher.TCode(tds, "levele 2> /dev/null")
myc.addOutputFile(out)

# Run Sobol analysis
tsobol = Sensitivity.TSobol(tds, myc, 10000)
tsobol.computeIndexes(args.computeIdxOpt) # args.computeIdxOpt = "" or "nointermed"

ntresu = tsobol.getResultTuple()

# Plotting mess
colors = [1, 2, 3, 4, 6, 7, 8, 15, 30, 38, 41, 46]
tps = np.array([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],
               dtype=float)

c2 = ROOT.TCanvas("c2", "c2", 5, 64, 1200, 900)
pad = ROOT.TPad("pad", "pad", 0, 0.03, 1, 1)
pad.Draw()
pad.Divide(1, 2)
pad.cd(1)
ROOT.gPad.SetLogx()
ROOT.gPad.SetGrid()

# LegendandMArker
ROOT.gStyle.SetMarkerStyle(3)
ROOT.gStyle.SetLegendBorderSize(0)
ROOT.gStyle.SetFillStyle(0)

mg2 = ROOT.TMultiGraph()
names = ["t", "kl", "kc", "v1", "l1", "r1", "rc1",
         "v2", "l2", "r2", "rc2", "w"]
fdeg = [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]
leg = [0., 0., 0., 0., 0., 0.]
Zeros = np.zeros([26])

for igr in range(12):

    sel = "Inp==\""+names[igr]+"\""
    sel += " && Order==\"First\" &&  Algo==\"martinez11\""
    ntresu.Draw("CILower:CIUpper:Value", sel, "goff")
    data = ntresu.GetV3()
    themin = ntresu.GetV1()
    themax = ntresu.GetV2()
```

(continues on next page)

(continued from previous page)

```
    for i in range(26):
        themin[i] = data[i] - themin[i]
        themax[i] = - data[i] + themax[i]

    if (igr % 2) == 0:
        leg[int(igr/2)] = ROOT.TLegend(0.1+0.15*(igr/2), 0.91,
                                       0.25+0.15*(igr/2), 0.98)
        leg[int(igr/2)].SetTextSize(0.045)

    fdeg[igr] = ROOT.TGraphAsymmErrors(26, tps, data,
                                       Zeros, Zeros, themin, themax)
    fdeg[igr].SetMarkerColor(colors[igr])
    fdeg[igr].SetLineColor(colors[igr])
    fdeg[igr].SetFillColor(colors[igr])
    leg[int(igr/2)].AddEntry(fdeg[igr], names[igr], "pl")

for igr2 in range(11, -1, -1):
    mg2.Add(fdeg[igr2])

mg2.Draw("APC")
mg2.GetXaxis().SetTitle("Time")
mg2.GetYaxis().SetTitle("S_{1}[martinez11]")
for igr3 in range(6):
    leg[igr3].Draw()

mg = ROOT.TMultiGraph()
tdeg = [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]
pad.cd(2)
ROOT.gPad.SetLogx()
ROOT.gPad.SetGrid()
for igr in range(12):

    sel = "Inp==\""+names[igr]+"\""
    sel += " && Order==\"Total\" &&  Algo==\"martinez11\""
    ntresu.Draw("CILower:CIUpper:Value", sel, "goff")
    data = ntresu.GetV3()
    themin = ntresu.GetV1()
    themax = ntresu.GetV2()
    for i in range(26):
        themin[i] = data[i] - themin[i]
        themax[i] = - data[i] + themax[i]

    for ip in range(26):
        if ip == 0:
            tdeg[igr] = ROOT.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])
```

(continues on next page)

(continued from previous page)

```
for igr2 in range(11, -1, -1):
    mg.Add(tdeg[igr2])

mg.Draw("APC")
mg.GetXaxis().SetTitle("Time")
mg.GetYaxis().SetTitle("S_{T}[martinez11]")
for igr3 in range(6):
    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(DS.TAttribute("y", DS.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).

13.6.12.3 Graph

The results of the previous macro is shown in Figure 13.28, 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.

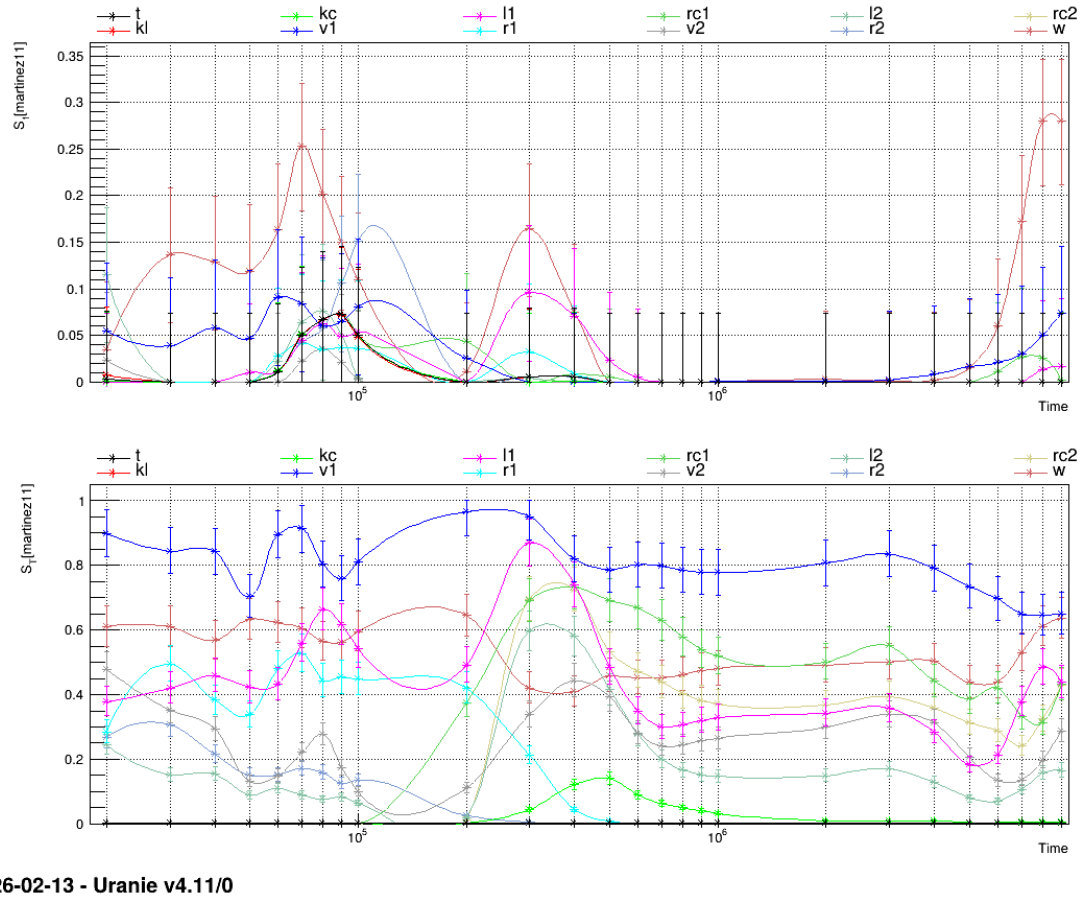


Figure 13.28: Graph of the macro **"sensitivitySobolLeveLE.py"**

13.6.13 Macro "sensitivitySobolRe-estimation.py"

13.6.13.1 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 *Computation of Sobol's sensitivity indices* 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 *Macro "sensitivitySobolFunctionFlowrate.py"*.

13.6.13.2 Macro Uranie

```
"""
Example of sobol re-estimation
"""
```

(continues on next page)

(continued from previous page)

```
from URANIE import DataServer, Sensitivity
import ROOT

ROOT.gROOT.LoadMacro("UserFunctions.C")

# Define the DataServer
tds = DataServer.TDataServer("tdsflowreate", "DataBase flowreate")
tds.fileDataRead("ref_sobol_launching_.dat")

tsobol = Sensitivity.TSobol(tds, "rw:r:tu:tl:hu:hl:l:kw", "flowrateModel")
tsobol.setDrawProgressBar(False)
tsobol.computeIndexes()

cc = ROOT.TCanvas("c1", "histgramme", 5, 64, 1270, 667)
pad = ROOT.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.py`. 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 = Sensitivity.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 *Console* from where the full sample is coming from.

13.6.13.3 Graph

The results of the previous macro is shown in Figure 13.29, 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.

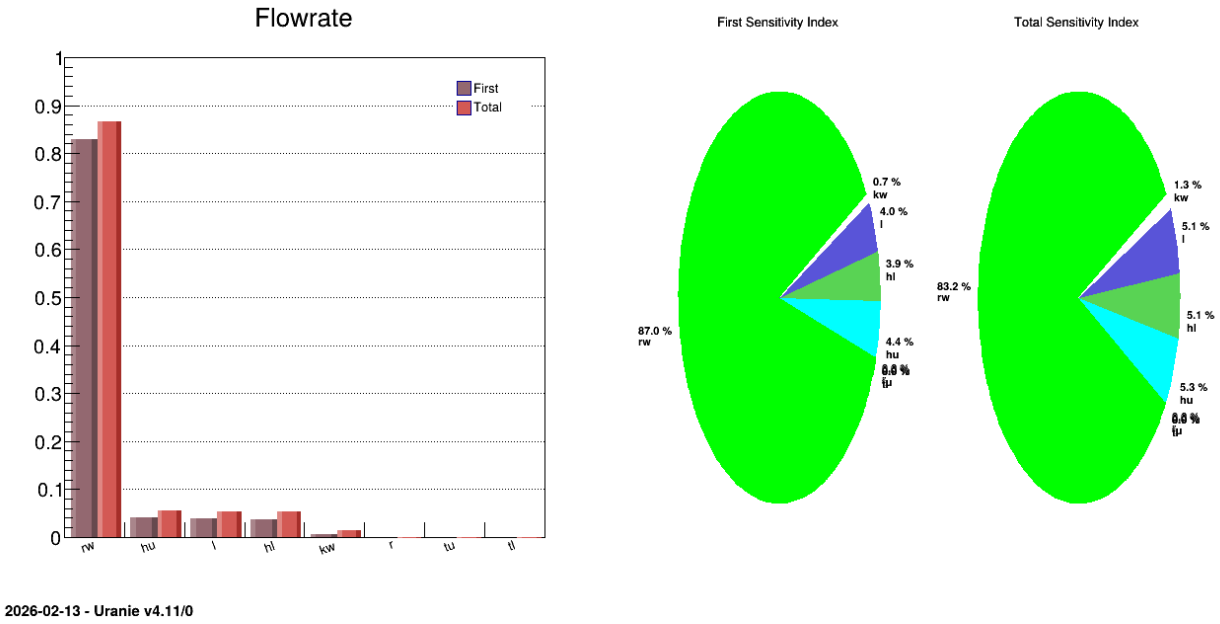


Figure 13.29: Graph of the macro **"sensitivitySobolRe-estimation.py"**

13.6.13.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

 ** 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]
```

13.6.14 Macro "sensitivitySobolWithData.py"

13.6.14.1 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 *Computation of Sobol's sensitivity indices* 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 *Macro "sensitivitySobolFunctionFlowrate.py"*.

13.6.14.2 Macro Uranie

```
"""
Example of sobol estimation using provided data
"""
from URANIE import DataServer, Sensitivity
import ROOT

ROOT.gROOT.LoadMacro("UserFunctions.C")
# Define the DataServer
tds = DataServer.TDataServer("tdsflowreate", "DataBase flowrate")
tds.fileDataRead("_onlyMandN_sobol_launching_.dat")

ns = 10000
tsobol = Sensitivity.TSobol(tds, "flowrateModel", ns, "rw:r:tu:tl:hu:hl:l:kw",
                            "flowrateModel", "WithData")
tsobol.setDrawProgressBar(False)
tsobol.computeIndexes()

cc = ROOT.TCanvas("c1", "histgramme", 5, 64, 1270, 667)
pad = ROOT.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 *Macro "sensitivitySobolFunctionFlowrate.py"*, the `UserFunctions.C` file is loaded and the input file `_onlyMandN_sobol_launching_.dat` (which should have been generated by the use of `sensitivtySobolFunctionFlowrate.py`), 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 = Sensitivity.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 *Console* from where the original set of points is coming from.

13.6.14.3 Graph

The results of the previous macro is shown in Figure 13.30, 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.

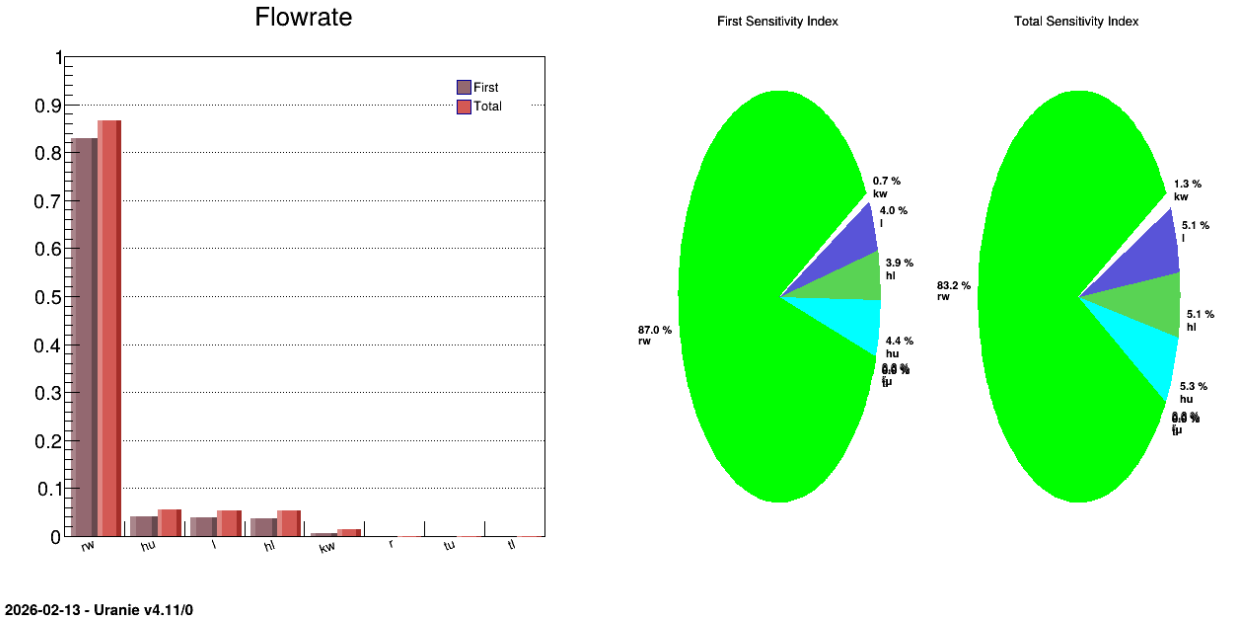


Figure 13.30: Graph of the macro **"sensitivitySobolWithData.py"**

13.6.14.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

 ** 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]
```

13.6.15 Macro "sensitivitySobolLoadFile.py"

13.6.15.1 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, be 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 *Computation of Sobol's sensitivity indices*.

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 *Macro "sensitivitySobolFunctionFlowrate.py"*.

13.6.15.2 Macro Uranie

```
"""
Example of sobol estimation once a file is loaded
"""
from URANIE import DataServer, Sensitivity
import ROOT

ROOT.gROOT.LoadMacro("UserFunctions.C")

# Define the DataServer
tds = DataServer.TDataServer("tdsflowreate", "DataBase flowreate")
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))

ns = 100000
tsobol = Sensitivity.TSobol(tds, "flowrateModel", ns,
                            "rw:r:tu:tl:hu:hl:l:kw",
                            "flowrateModel", "DummyPython")
tsobol.loadOtherSobolFile("ref_sobol_launching_.dat")
tsobol.setDrawProgressBar(False)
tsobol.computeIndexes()

myfilter = "Algo==\"--first--\" || Algo==\"--total--\""
tsobol.getResultTuple().Scan("*", myfilter)

cc = ROOT.TCanvas("c1", "histgramme", 5, 64, 1270, 667)
pad = ROOT.TPad("pad", "pad", 0, 0.03, 1, 1)
```

(continues on next page)

(continued from previous page)

```
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 *Macro "sensitivitySobolFunctionFlowrate.py"*, 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 *Macro "sensitivitySobolFunctionFlowrate.py"* which provided the input file that would be used to complete this estimation).

All the code lines are exactly those taken out of *Macro Uranie* 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 *Console* to *Console*.

13.6.15.3 Graph

The results of the previous macro is shown in Figure 13.31, 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.

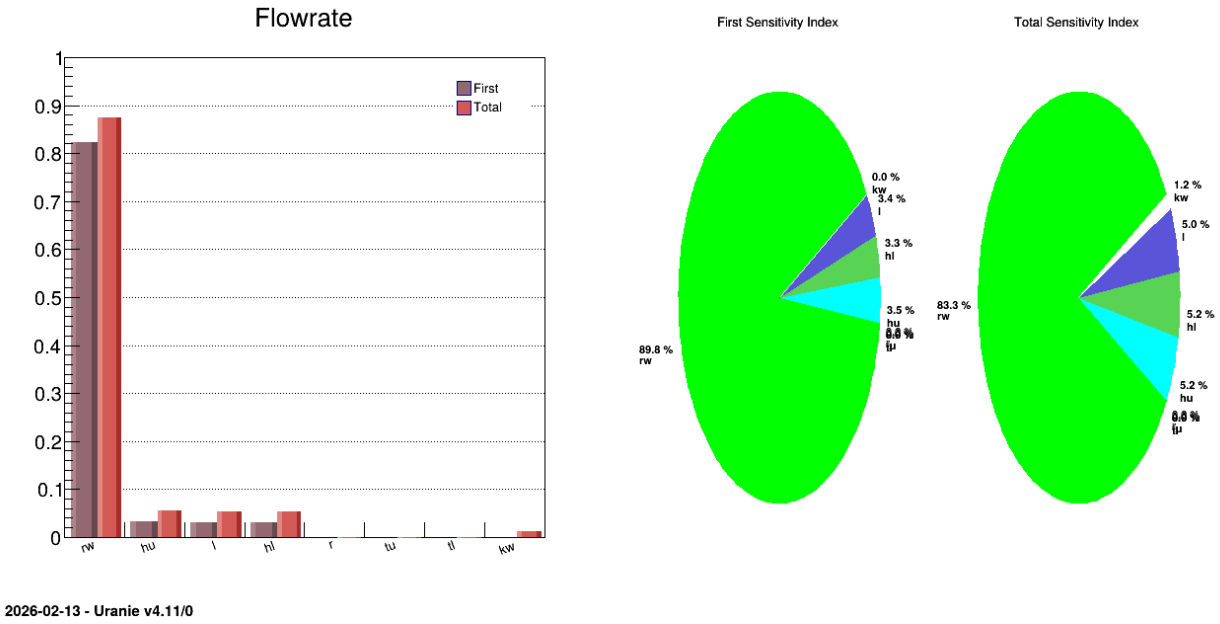


Figure 13.31: Graph of the macro **"sensitivitySobolLoadFile.py"**

13.6.15.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[760]
 <URANIE::WARNING> TDataServer::fileDataRead: Expected iterator tdsflowreate_1__n__iter__ not found but tdsflowreate__n_
↪_iter__ looks like an URANIE iterator => Will be used as so.
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8531]
 <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 *
*****************************************************************************************************
```

(continues on next page)

(continued from previous page)

```
*        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
```

13.6.16 Macro "sensitivityJohnsonRWFunctionFlowrate.py"

13.6.16.1 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 *Presentation of the problem*.

13.6.16.2 Macro Uranie

```
"""
Example of Johnson relative weight method applied to flowrate
"""
from URANIE import DataServer, Sensitivity
import ROOT

ROOT.gROOT.LoadMacro("UserFunctions.C")

# Define the DataServer
tds = DataServer.TDataServer("tdsflowrate", "DataBase flowrate")
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))

# param Size of a sampling.
nS = 1000
FuncName = "flowrateModel"

tjrw = Sensitivity.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
cc = ROOT.TCanvas("canhist", "histgramme")
tjrw.drawIndexes("Flowrate", "", "nonewcanv,hist,first")
cc.Print("appliUranieFlowrateJohnsonRW1000Histogram_py.png")

ccc = ROOT.TCanvas("canpie", "TPie")
tjrw.drawIndexes("Flowrate", "", "nonewcanv,pie")
ccc.Print("appliUranieFlowrateJohnsonRW1000Pie_py.png")
```

The function `flowrateModel` is loaded from the macro `UserFunctions.C` (the file can be found in `${URANIESYS}/share/uranie/macros`)

```
ROOT.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
tds = DataServer.TDataServer("tdsflowrate", "DataBase flowrate")
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.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:

```
tjrw = Sensitivity.TJohnsonRW(tds, FuncName, nS,
                              "rw:r:tu:tl:hu:hl:l:kw", FuncName)
```

Computation of sensitivity indexes:

```
tjrw.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.

13.6.16.3 Graph

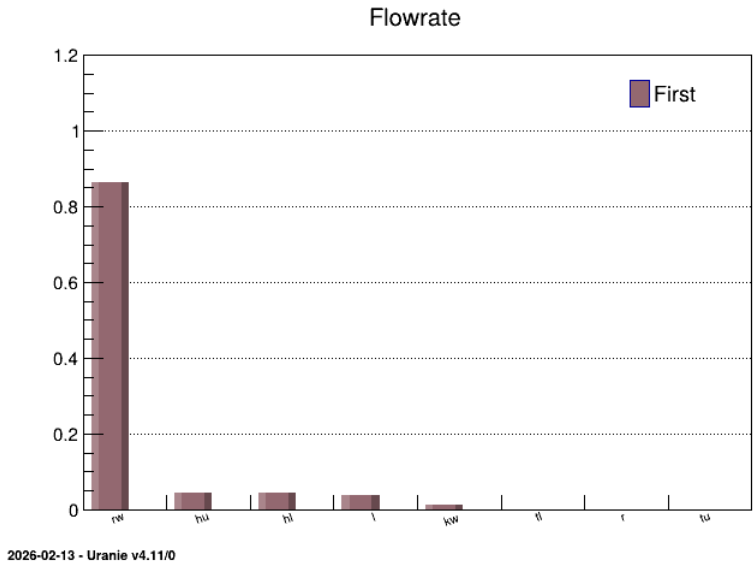


Figure 13.32: Graph of the macro **"sensitivityJohnsonRWFunctionFlowrate.py"**

13.6.16.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026
```

(continues on next page)

(continued from previous page)

```
 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8531]
 <URANIE::WARNING> TDataServer::getTuple Error : There is no tree!
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${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
```

13.6.17 Macro “sensitivityJohnsonRWCorrelatedFunctionFlowrate.py”

13.6.17.1 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 *Presentation of the problem*. Compared to version detailled in *Macro “sensitivityJohnsonRWFunctionFlowrate.py”*, the idea is here to correlate the input variables with a random correlation matrix.

13.6.17.2 Macro Uranie

```
"""
Example of Johson relative weight method applied to flowrate
"""
from math import sqrt
from URANIE import DataServer, Sensitivity
import ROOT

def gen_corr(_nx=8, correlated=True, seed=0):
    """Generate randomly a good, highly-correlated, input correlation matrix."""
    # Define a randomly filled matrix
```

(continues on next page)

(continued from previous page)

```
    a_mat = ROOT.TMatrixD(_nx, _nx)
    for i in range(_nx):
        for j in range(_nx):
            a_mat[i][j] = ROOT.gRandom.Gaus(0, 1)

    # Compute AA^T and normalise it to get "covariance matrix"
    gamma = ROOT.TMatrixD(a_mat, ROOT.TMatrixD.kMultTranspose, a_mat)
    gamma *= 1. / _nx

    # Inverse of the diagonal matrix to do as if this was 1/sqrt(variance)
    sig = ROOT.TMatrixD(_nx, _nx)
    for i in range(_nx):
        sig[i][i] = 1. / sqrt(gamma[i][i])

    # Compute the input correlation matrix
    in_corr_right = ROOT.TMatrixD(gamma, ROOT.TMatrixD.kMult, sig)
    in_corr = ROOT.TMatrixD(sig, ROOT.TMatrixD.kMult, in_corr_right)
    if not correlated:
        in_corr.UnitMatrix()

    return in_corr


ROOT.gROOT.LoadMacro("UserFunctions.C")

# Define the DataServer
tds = DataServer.TDataServer("tdsflowrate", "DataBase flowrate")
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))

# \param Size of a sampling.
nS = 1000
FuncName = "flowrateModel"

# Get a correlation matrix for the inputs
inCorr = gen_corr(8, True, args.seed)
inCorr.Print()

tjrw = Sensitivity.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\"")
```

(continues on next page)

(continued from previous page)

```
# Get the results as plots
cc = ROOT.TCanvas("canhist", "histgramme")
tjrw.drawIndexes("Flowrate", "", "nonewcanv, hist, first")
cc.Print("appliUranieFlowrateJohnsonRWCorrelated1000Histogram_py.png")

ccc = ROOT.TCanvas("canpie", "TPie")
tjrw.drawIndexes("Flowrate", "", "nonewcanv, pie")
ccc.Print("appliUranieFlowrateJohnsonRWCorrelated1000Pie_py.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}/share/uranie/macros`)

```
ROOT.gROOT.LoadMacro("UserFunctions.C")
```

Compared to the discussion in *Macro Uranie*, 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
inCorr = gen_corr(8, True, args.seed)
tjrw = Sensitivity.TJohnsonRW(tds, FuncName, nS,
                              "rw:r:tu:tl:hu:hl:l:kw", FuncName)
# Set the correlation
tjrw.setInputCorrelationMatrix(inCorr)
```

The computation of sensitivity indices can finally be done:

```
tjrw.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.

13.6.17.3 Graph

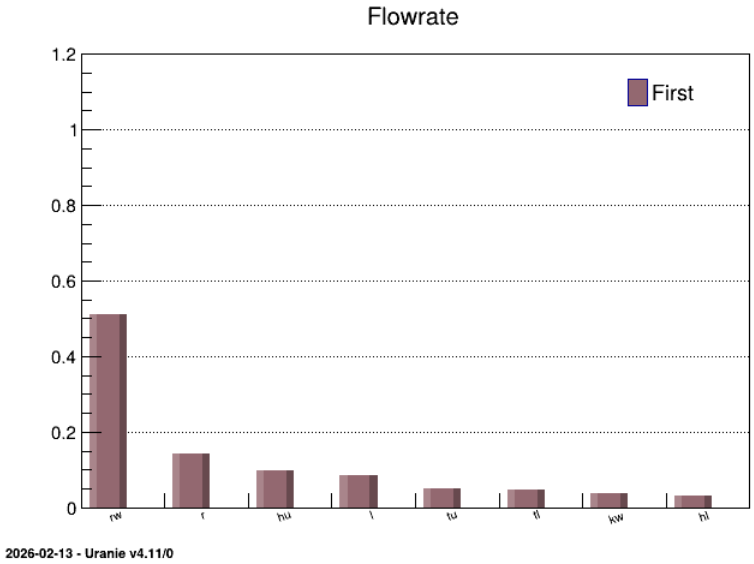


Figure 13.33: Graph of the macro **"sensitivityJohnsonRWCorrelatedFunctionFlowrate.py"**

13.6.17.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026


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
```

(continues on next page)

(continued from previous page)

```
   6 |    -0.1943           1     -0.4415
   7 |     0.2848     -0.4415           1

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[8531]
 <URANIE::WARNING> TDataServer::getTuple Error : There is no tree!
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${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
```

13.6.18 Macro "sensitivityJohnsonRWJustCorrelationFakeFlowrate.py"

13.6.18.1 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 *Macro "sensitivityJohnsonRWCorrelatedFunctionFlowrate.py"*, 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.

13.6.18.2 Macro Uranie

```
"""
Example of Johnson relative weight applied only to a correlation matrix
"""
from math import sqrt
from URANIE import DataServer, Sensitivity
```

(continues on next page)

(continued from previous page)

```
import ROOT

def gen_corr(_nx=8, correlated=True, seed=0):
    """Generate randomly a good, highly-correlated, correlation matrix for inputs
    define the proper covariance with the output to do as if this output if a
    perfect linear combination of the inputs."""
    # Define a randomly filled matrix
    a_mat = ROOT.TMatrixD(_nx, _nx)
    for i in range(_nx):
        for j in range(_nx):
            a_mat[i][j] = ROOT.gRandom.Gaus(0, 1)

    # Compute AA^T and normalise it to get "covariance matrix"
    gamma = ROOT.TMatrixD(a_mat, ROOT.TMatrixD.kMultTranspose, a_mat)
    gamma *= 1. / _nx

    # Inverse of the diagonal matrix to do as if this was 1/sqrt(variance)
    sig = ROOT.TMatrixD(_nx, _nx)
    for i in range(_nx):
        sig[i][i] = 1. / sqrt(gamma[i][i])

    # Compute the input correlation matrix
    in_corr_right = ROOT.TMatrixD(gamma, ROOT.TMatrixD.kMult, sig)
    in_corr = ROOT.TMatrixD(sig, ROOT.TMatrixD.kMult, in_corr_right)
    if not correlated:
        in_corr.UnitMatrix()
    var_y = in_corr.Sum()

    # Proper correlation, output included
    corr = ROOT.TMatrixD(_nx+1, _nx+1)
    corr.UnitMatrix()
    # putting already defined input
    corr.SetSub(0, 0, in_corr)
    # Adjust the covariance of the output wrt to all inputs
    for i in range(_nx):
        value = 0
        for j in range(_nx):
            value += in_corr[i][j]

        corr[_nx][i] = corr[i][_nx] = value / sqrt(var_y)

    return corr


# Define the DataServer
tds = DataServer.TDataServer("tdsflowrate", "DataBase flowrate")
tds.addAttribute("rw")
tds.addAttribute("r")
tds.addAttribute("tu")
tds.addAttribute("tl")
tds.addAttribute("hu")
tds.addAttribute("hl")
```

(continues on next page)

(continued from previous page)

```
tds.addAttribute("l")
tds.addAttribute("kw")
# outputs
tds.addAttribute("flowrateModel")
tds.getAttribute("flowrateModel").setOutput()


# Get the full correlation matrix
inCorr = gen_corr(8, True, args.seed)

# Johnson definition
tjrw = Sensitivity.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
cc = ROOT.TCanvas("canhist", "histgramme")
tjrw.drawIndexes("Flowrate", "", "nonewcanv, hist, first")
cc.Print("appliUranieFakeFlowrateJohnsonRWCorrelationHistogram_py.png")

ccc = ROOT.TCanvas("canpie", "TPie")
tjrw.drawIndexes("Flowrate", "", "nonewcanv, pie")
ccc.Print("appliUranieFakeFlowrateJohnsonRWCorrelationPie_py.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
tds = DataServer.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 *Macro Uranie*, the only differences are the instantiation of the `TJohnsonRW` object and the method code to provide the correlation matrix.

```
# Get the full correlation matrix
inCorr = gen_corr(8, True, args.seed)

# Johnson definition
tjrw = Sensitivity.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 *Macro Uranie* which is `setInputCorrelationMatrix`, which set only the input correlation matrix.

The computation of sensitivity indices can finally be done:

```
tjrw.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.

13.6.18.3 Graph

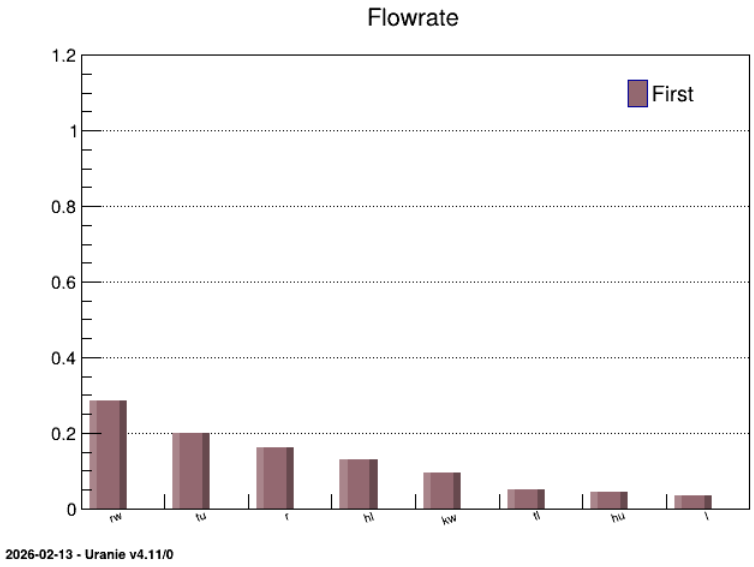


Figure 13.34: Graph of the macro **"sensitivityJohnsonRWJustCorrelationFakeFlowrate.py"**

13.6.18.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[2202]
```

(continues on next page)

(continued from previous page)

```
 <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[8531]
 <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
```

13.6.19 Macro “sensitivityHSICFunctionFlowrate.py”

13.6.19.1 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 *Presentation of the problem*.

13.6.19.2 Macro Uranie

```
"""
Example of HSIC method applied to flowrate
"""
from URANIE import DataServer, Sensitivity, Launcher, Sampler
import ROOT

ROOT.gROOT.LoadMacro("UserFunctions.C")

# Define the DataServer
tds = DataServer.TDataServer("tdsflowrate", "DataBase flowrate")
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
```

(continues on next page)

(continued from previous page)

```
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))

# Generation of the sample (it can be a given sample).
nS = 500
sampling = Sampler.TSampling(tds, "lhs", nS)
sampling.generateSample()

tlf = Launcher.TLauncherFunction(tds, "flowrateModel")
tlf.setDrawProgressBar(False)
tlf.run()

# Create a THSIC object, compute indexes and print results
thsic = Sensitivity.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","",
↪"colsize=5 col=6:8::9:8:8:8")

# Print HSIC indexes
can = ROOT.TCanvas("c1", "Graph 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/uranie/macros`)

```
ROOT.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
tds = DataServer.TDataServer("tdsflowrate", "DataBase flowrate")
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.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`

```
nS = 500
sampling = Sampler.TSampling(tds, "lhs", nS)
sampling.generateSample()
```

(continues on next page)

(continued from previous page)

```
tlf = Launcher.TLauncherFunction(tds, "flowrateModel")
tlf.setDrawProgressBar(False)
tlf.run()
```

To instantiate the `THSIC` object, one uses the `TDataServer`, the name of the input and output variables:

```
thsic = Sensitivity.THSIC(tds, "rw:r:tu:tl:hu:hl:l:kw","flowrateModel")
```

Computation of sensitivity indexes:

```
thsic.computeIndexes("quiet")
```

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.

13.6.19.3 Graph

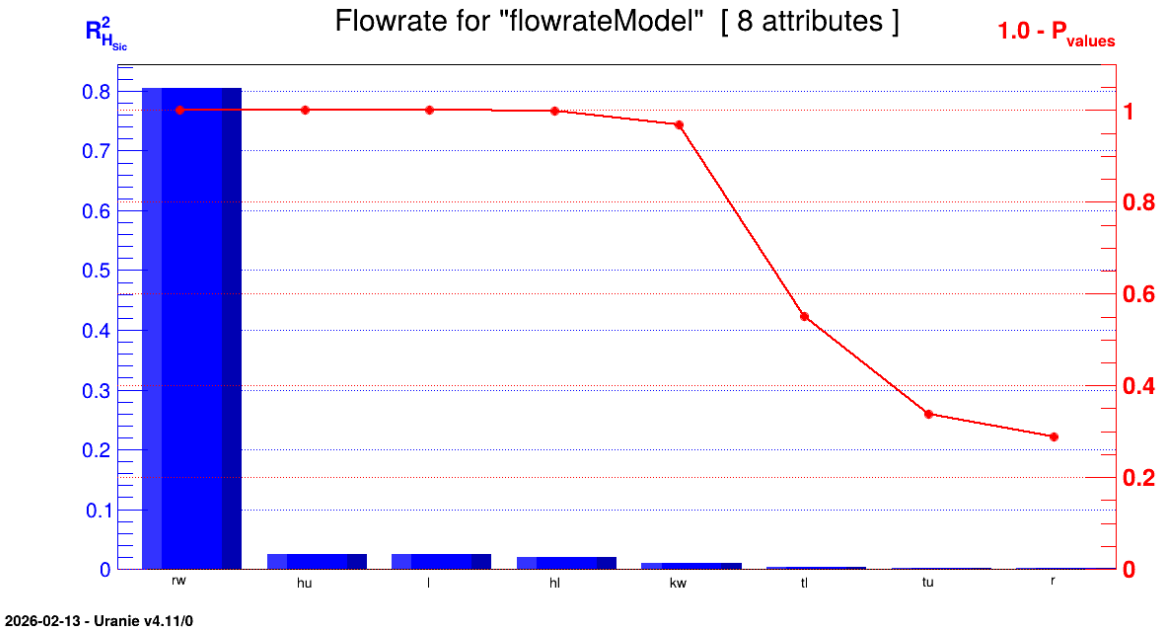


Figure 13.35: Graph of the macro **"sensitivityHSICFunctionFlowrate.py"**

13.6.19.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

*******************************************************************************
*    Row    *    Out *      Inp * Metho *     Order *    Value *  CILower *  CIUpper *
*******************************************************************************
*        0 * flowra *       rw *  HSic *    R2HSic * 0.804431 *       -1 *       -1 *
```

(continues on next page)

(continued from previous page)

```
*        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 *
*************************************************************************************
```

13.6.20 Macro “sensitivitySobolRankFunctionFlowrate.py”

13.6.20.1 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 *Presentation of the problem*.

13.6.20.2 Macro Uranie

```
"""
Example of HSIC method applied to flowrate
"""
from URANIE import DataServer, Sensitivity, Launcher, Sampler
import ROOT

ROOT.gROOT.LoadMacro("UserFunctions.C")

# Define the DataServer
tds = DataServer.TDataServer("tdsflowrate", "DataBase flowrate")
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))
```

(continues on next page)

(continued from previous page)

```
# Generation of the sample (it can be a given sample).
nS = 500
sampling = Sampler.TSampling(tds, "lhs", nS)
sampling.generateSample()

tlf = Launcher.TLauncherFunction(tds, "flowrateModel")
tlf.setDrawProgressBar(False)
tlf.run()

# Create a TSobolRank object, compute indexes and print results
tsobolrank = Sensitivity.TSobolRank(tds, "rw:r:tu:tl:hu:hl:l:kw","flowrateModel")
tsobolrank.computeIndexes()
tsobolrank.getResultTuple().SetScanField(60)
tsobolrank.getResultTuple().Scan("Out:Inp:Method:Order:Value:CILower:CIUpper","",
↪"colsize=5 col=6:8::9:8:8:8")

# Print Sobol indexes
can = ROOT.TCanvas("c1", "Graph 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/uranie/macros`)

```
ROOT.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
tds = DataServer.TDataServer("tdsflowrate", "DataBase flowrate")
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.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`

```
nS = 500
sampling = Sampler.TSampling(tds, "lhs", nS)
sampling.generateSample()

tlf = Launcher.TLauncherFunction(tds, "flowrateModel")
tlf.setDrawProgressBar(False)
tlf.run()
```

To instantiate the `TSobolRank` object, one uses the `TDataServer`, the name of the input and output variables:

```
tsobolrank = Sensitivity.TSobolRank(tds, "rw:r:tu:tl:hu:hl:l:kw","flowrateModel")
```

Computation of sensitivity indexes:

```
tsobolrank.computeIndexes()
```

It will produce one plot containing the first Sobol indices. The console is also shown below for completness.

13.6.20.3 Graph

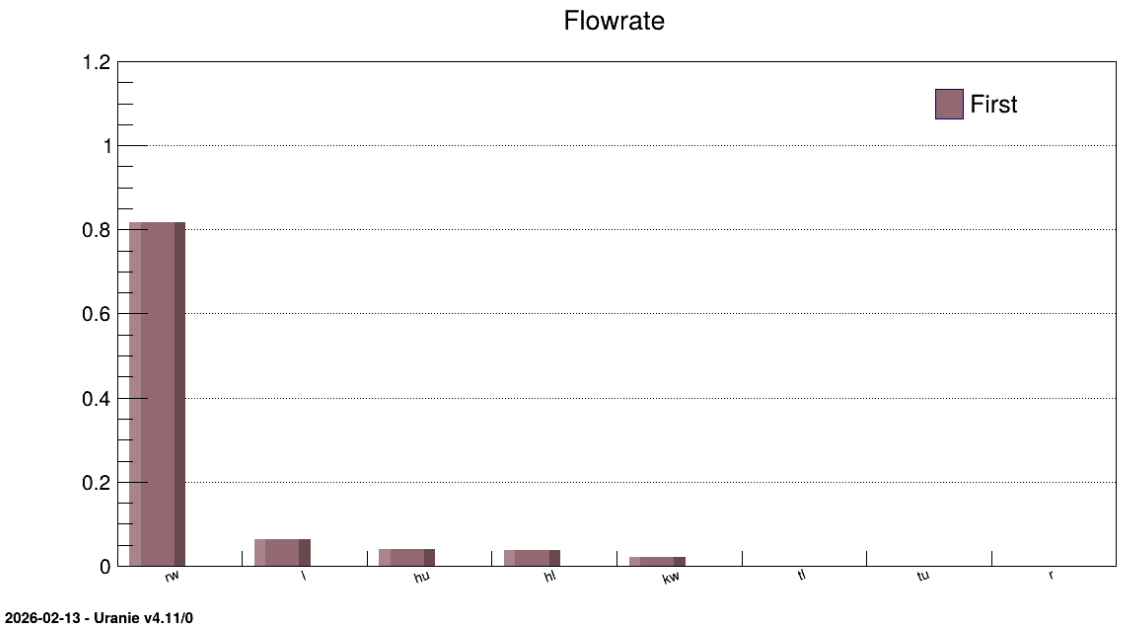


Figure 13.36: Graph of the macro **"sensitivitySobolRankFunctionFlowrate.py"**

13.6.20.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

*********************************************************************************
*    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 *
*********************************************************************************
```

13.7 Macros Modeler

13.7.1 Macro “modelerCornellLinearRegression.py”

13.7.1.1 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
```

13.7.1.2 Macro Uranie

```
"""
Example of linear regression model
"""
from URANIE import DataServer, Modeler
import ROOT

tds = DataServer.TDataServer()
tds.fileDataRead("cornell.dat")

tds.normalize("*", "cr")  # "*" is equivalent to "x1:x2:x3:x4:x5:x6:y"
# Third field not filled implies DataServer.TDataServer.kCR

c = ROOT.TCanvas("c1", "Graph for the Macro modeler", 5, 64, 1270, 667)
pad = ROOT.TPad("pad", "pad", 0, 0.03, 1, 1)
pad.Draw()
pad.Divide(2)
pad.cd(1)
tds.draw("y:x7")

tlin = Modeler.TLinearRegression(tds, "x1cr:x2cr:x3cr:x4cr:x5cr:x6cr",
                                 "ycr", "nointercept")
tlin.estimate()

pad.cd(2)
```

(continues on next page)

(continued from previous page)

```
tds.draw("ycr:ycrhat")
```

The ASCII data file `cornell.dat` is loaded in the `TDataServer`:

```
tds = DataServer.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")  # "*" is equivalent to "x1:x2:x3:x4:x5:x6:y"
```

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:

```
tlin = Modeler.TLinearRegression(tds, "x1cr:x2cr:x3cr:x4cr:x5cr:x6cr",
                                 "ycr", "nointercept")
tlin.estimate()
```

13.7.1.3 Graph

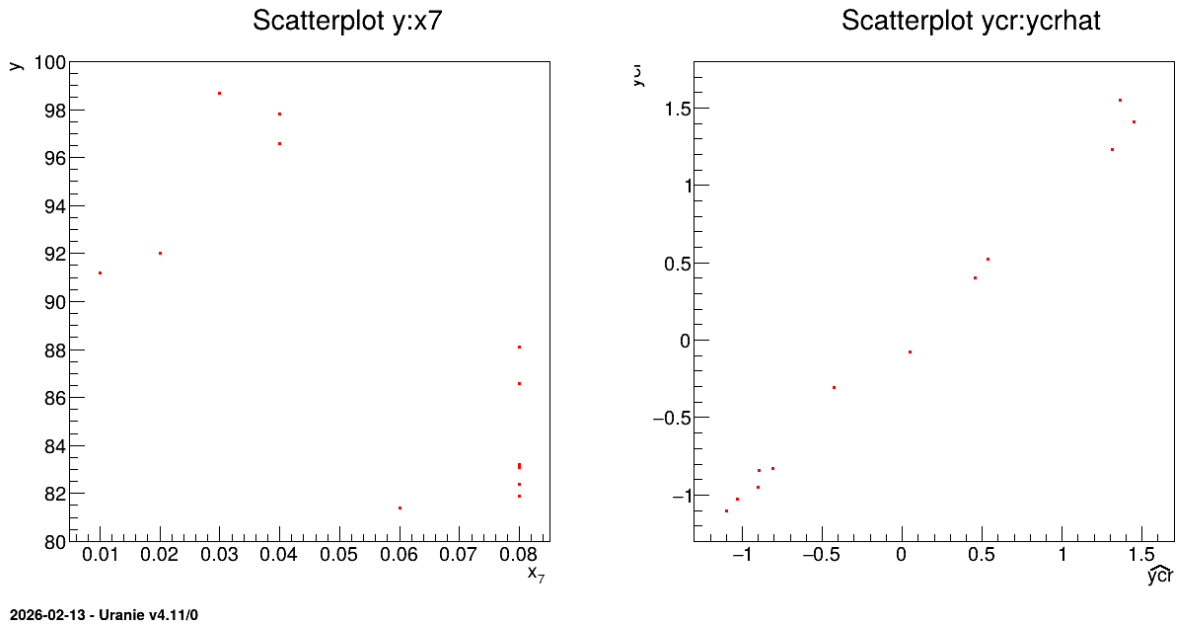


Figure 13.37: Graph of the macro **"modelerCornellLinearRegression.py"**

13.7.2 Macro "modelerFlowrateLinearRegression.py"

13.7.2.1 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 *Presentation of the problem* on 500 patterns.

13.7.2.2 Macro Uranie

```
"""
Example of linear regression on flowrate
"""
from URANIE import DataServer, Modeler
import ROOT

tds = DataServer.TDataServer()
tds.fileDataRead("flowrate_sampler_launcher_500.dat")

c = ROOT.TCanvas("c1", "Graph for the Macro modeler", 5, 64, 1270, 667)
pad = ROOT.TPad("pad", "pad", 0, 0.03, 1, 1)
pad.Draw()
pad.Divide(2)
pad.cd(1)
tds.draw("yhat:rw")

# tds.getAttribute("yhat").setOutput()

tlin = Modeler.TLinearRegression(tds, "rw", "yhat", "DummyForPython")
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:

```
tds = DataServer.TDataServer()
tds.fileDataRead("flowrate_sampler_launcher_500.dat")
```

The linear regression is initialised and characteristic values are computed for **rw** with the `estimate` method:

```
tlin = Modeler.TLinearRegression(tds, "rw", "yhat", "DummyForPython")
tlin.estimate()
```

The last argument is the option field, which in most cases is empty. Here it is filled with "DummyPython" which helps specify to python which constructor to choose. There are indeed several possible constructors these 5 five first arguments, but C++ can make the difference between them as the literal members are either `std::string`, `ROOT::TString`, `Char_t*` or even `Option_t*`. For python, these format are all `PyString`, so the sixth argument is compulsory to disentangle the possibilities.

13.7.2.3 Graph

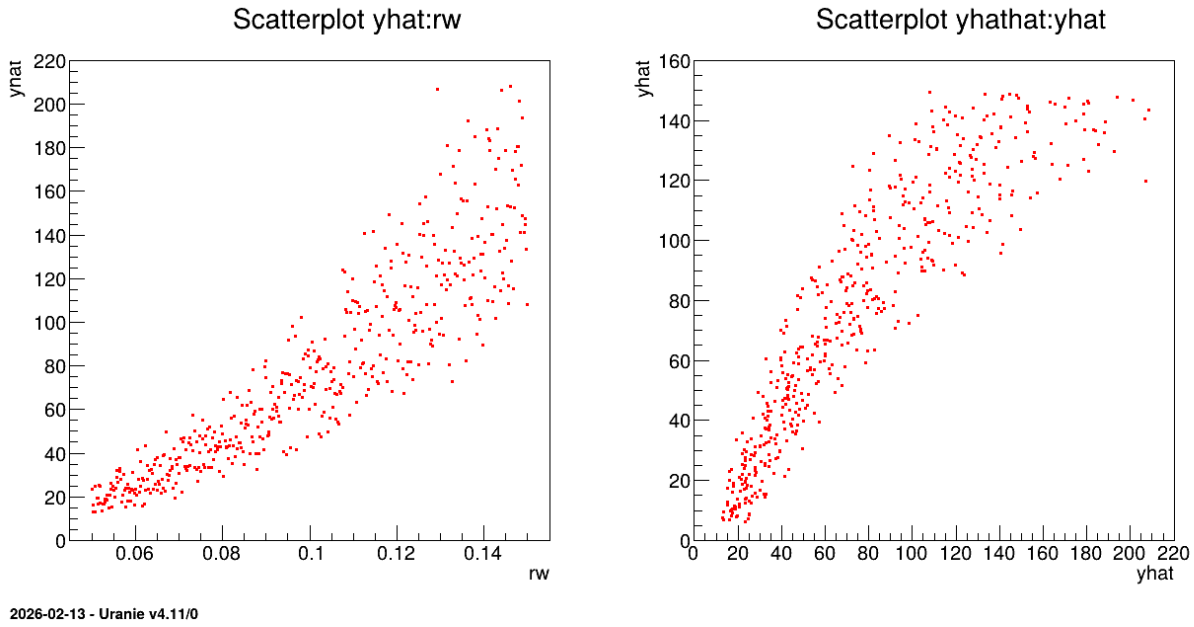


Figure 13.38: Graph of the macro **"modelerFlowrateLinearRegression.py"**

13.7.3 Macro "modelerFlowrateMultiLinearRegression.py"

13.7.3.1 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 *Presentation of the problem* on 500 patterns. Parameters of the regression are then exported in a .py 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
  //     ******************************************
```

(continues on next page)

(continued from previous page)

```
//
//     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
```

(continues on next page)

(continued from previous page)

```
  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.

13.7.3.2 Macro Uranie

```
"""
Example of multi-linear regression on flowrate
"""
from URANIE import DataServer, Launcher, Modeler, Sampler
import ROOT

tds = DataServer.TDataServer()
tds.fileDataRead("flowrate_sampler_launcher_500.dat")

c = ROOT.TCanvas("c1", "Graph for the Macro modeler", 5, 64, 1270, 667)

tlin = Modeler.TLinearRegression(tds, "rw:r:tu:tl:hu:hl:l:kw",
                                 "yhat", "DummyForPython")
tlin.estimate()

tlin.exportFunction("c++", "_FileContainingFunction_", "MultiLinearFunction")

# Define the DataServer
tds2 = DataServer.TDataServer("tdsFlowrate", "Design of experiments")

# Add the study attributes
tds2.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds2.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds2.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds2.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds2.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds2.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds2.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds2.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))

nS = 1000
# Generate DOE
sampling = Sampler.TSampling(tds2, "lhs", nS)
sampling.generateSample()

ROOT.gROOT.LoadMacro("_FileContainingFunction_.C")

# Create a TLauncherFunction from a TDataServer and an analytical function
# Rename the outpout attribute "ymod"
tlf = Launcher.TLauncherFunction(tds2, "MultiLinearFunction", "", "ymod")
# Evaluate the function on all the design of experiments
tlf.run()
```

(continues on next page)

(continued from previous page)

```
tds2.getTuple().SetMarkerColor(ROOT.kBlue)
tds2.getTuple().SetMarkerStyle(8)
tds2.getTuple().SetMarkerSize(1.2)

tds.getTuple().SetMarkerColor(ROOT.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:

```
tds = DataServer.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:

```
tlin = Modeler.TLinearRegression(tds, "rw:r:tu:tl:hu:hl:l:kw",
                                 "yhat", "DummyForPython")
tlin.estimate()
```

The last argument is the option field, which in most cases is empty. Here it is filled with "DummyPython" which helps specify to python which constructor to choose. There are indeed several possible constructors these 5 five first arguments, but C++ can make the difference between them as the literal members are either `std::string`, `ROOT::TString`, `Char_t*` or even `Option_t*`. For python, these format are all `PyString`, so the sixth argument is compulsory to disentangle the possibilities.

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(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds2.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds2.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds2.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds2.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds2.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds2.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds2.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))
```

A sampling is realised with a LHS method on 1000 patterns:

```
sampling = Sampler.TSampling(tds2, "lhs", nS)
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:

```
ROOT.gROOT.LoadMacro("_FileContainingFunction_.C")
tlf = Launcher.TLauncherFunction(tds2, "MultiLinearFunction", "", "ymod")
tlf.run()
```

13.7.3.3 Graph

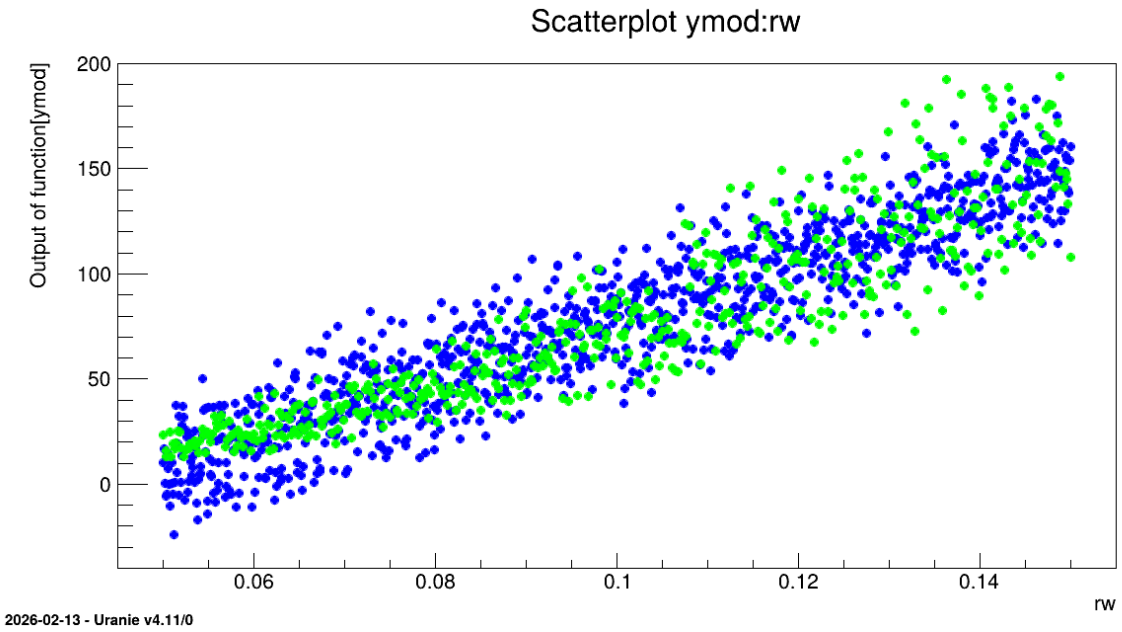


Figure 13.39: Graph of the macro **"modelerFlowrateMultiLinearRegression.py"**

13.7.4 Macro "modelerFlowrateNeuralNetworks.py"

13.7.4.1 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 *Presentation of the problem* 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.

13.7.4.2 Macro Uranie

```
"""
Example of neural network usage on flowrate
"""
from URANIE import DataServer, Launcher, Modeler
import ROOT

# Create a DataServer.TDataServer
# Load a database in an ASCII file
tdsu = DataServer.TDataServer("tdsu", "tds u")
```

(continues on next page)

(continued from previous page)

```
tdsu.fileDataRead("flowrate_sampler_launcher_500.dat")

#
tdsu.exportData("_flowrate_sampler_launcher_app_.dat", "",
                "tdsFlowrate__n__iter__<=300")
tdsu.exportData("_flowrate_sampler_launcher_val_.dat", "",
                "tdsFlowrate__n__iter__>300")

tds = DataServer.TDataServer("tdsFlowrate", "tds for flowrate")
tds.fileDataRead("_flowrate_sampler_launcher_app_.dat")

c = ROOT.TCanvas("c1", "Graph for the Macro modeler", 5, 64, 1270, 667)
# Builds a surrogate model (Artificial Neural Networks) from the DataBase
tann = Modeler.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")

ROOT.gROOT.LoadMacro("uranie_ann_flowrate.C")

tdsv = DataServer.TDataServer()
tdsv.fileDataRead("_flowrate_sampler_launcher_val_.dat")

print(tdsv.getNPatterns())

# evaluate the surrogate model on the database
tlf = Launcher.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`

```
tdsu = DataServer.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:

```
tds = DataServer.TDataServer("tdsFlowrate", "tds for flowrate")
tds.fileDataRead("_flowrate_sampler_launcher_app_.dat")
tann = Modeler.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","ANNflowrate")
```

The model is loaded from the macro `"uranie_ann_flowrate.C"` and applied on the second database with the function `ANNflowrate`:

```
ROOT.gROOT.LoadMacro("uranie_ann_flowrate.C")
tdsv = DataServer.TDataServer()
tdsv.fileDataRead("_flowrate_sampler_launcher_val_.dat")
tlf = Launcher.TLauncherFunction(tdsv, "ANNflowrate",
                                 "rw:r:tu:tl:hu:hl:l:kw", "yann")
tlf.run()
```

13.7.4.3 Graph

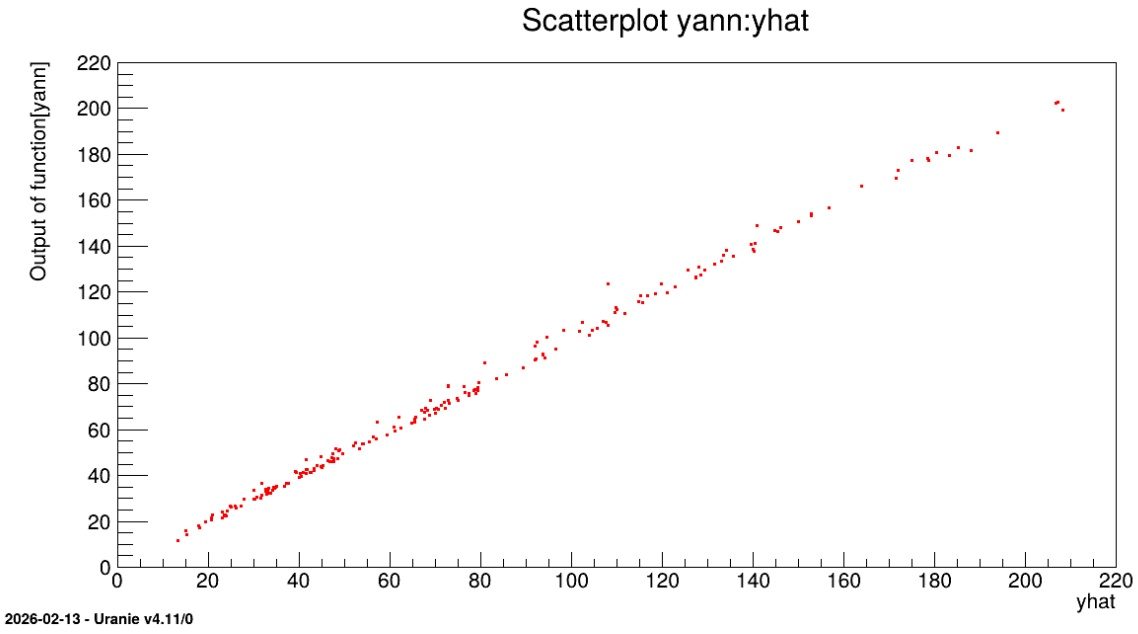


Figure 13.40: Graph of the macro **"modelerFlowrateNeuralNetworks.py"**

13.7.4.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
```

(continues on next page)

(continued from previous page)

```
                    Date: Thu Feb 12, 2026

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[760]
 <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: 1.84 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
******************************
```

13.7.5 Macro "modelerFlowrateNeuralNetworksLoadingPMML.py"

13.7.5.1 Objective

The objective of this macro is to build a surrogate model (an Artificial Neural Network) from a PMML file.

13.7.5.2 Macro Uranie

```
"""
Example of neural network loaded from pmml for flowrate
"""
from URANIE import DataServer, Launcher, Modeler
import ROOT

# Create a DataServer.TDataServer
# Load a database in an ASCII file
tds = DataServer.TDataServer("tdsFlowrate", "tds for flowrate")
tds.fileDataRead("_flowrate_sampler_launcher_app_.dat")

c = ROOT.TCanvas("c1", "Graph for the Macro modeler", 5, 64, 1270, 667)
# Build a surrogate model (Artificial Neural Networks) from the PMML file
tann = Modeler.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
ROOT.gROOT.LoadMacro("uranie_ann_flowrate_loaded.C")

tdsv = DataServer.TDataServer("tdsv", "tds for surrogate model")
tdsv.fileDataRead("_flowrate_sampler_launcher_val_.dat")

print(tdsv.getNPatterns())

# evaluate the surrogate model on the database
tlf = Launcher.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`:

```
tds = DataServer.TDataServer("tdsFlowrate", "tds for flowrate")
tds.fileDataRead("_flowrate_sampler_launcher_app_.dat")
```

The surrogate model is loaded from a PMML file:

```
tann = Modeler.TANNModeler(tds, "uranie_ann_flowrate.pmml", "ANNflowrate")
```

The model is exported in an external file in C++ language `"uranie_ann_flowrate_loaded.C"` where the function name is `ANNflowrate`:

```
tann.exportFunction("c++", "uranie_ann_flowrate_loaded", "ANNflowrate")
```

The model is loaded from the macro `"uranie_ann_flowrate_loaded.C"` and applied on the database with the function `ANNflowrate`:

```
ROOT.gROOT.LoadMacro("uranie_ann_flowrate_loaded.C")
tdsv = DataServer.TDataServer("tdsv", "tds for surrogate model")
tdsv.fileDataRead("_flowrate_sampler_launcher_val_.dat")
tlf = Launcher.TLauncherFunction(tdsv, "ANNflowrate",
                                 "rw:r:tu:tl:hu:hl:l:kw", "yann")
tlf.run()
```

13.7.5.3 Graph

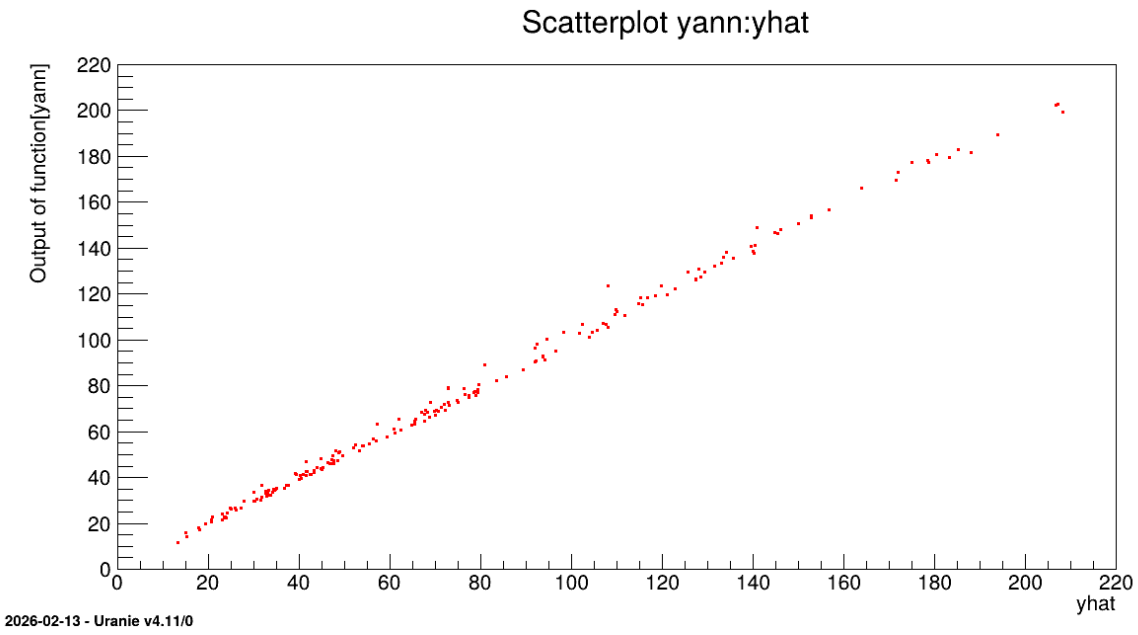


Figure 13.41: Graph of the macro **"modelerFlowrateNeuralNetworksLoadingPMML.py"**

13.7.5.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

PMML Constructor: uranie_ann_flowrate.pmml

*******************************
*** TModeler::exportFunction lang[c++] file[uranie_ann_flowrate_loaded]␣
↪name[ANNflowrate] soption[]

*******************************
```

(continues on next page)

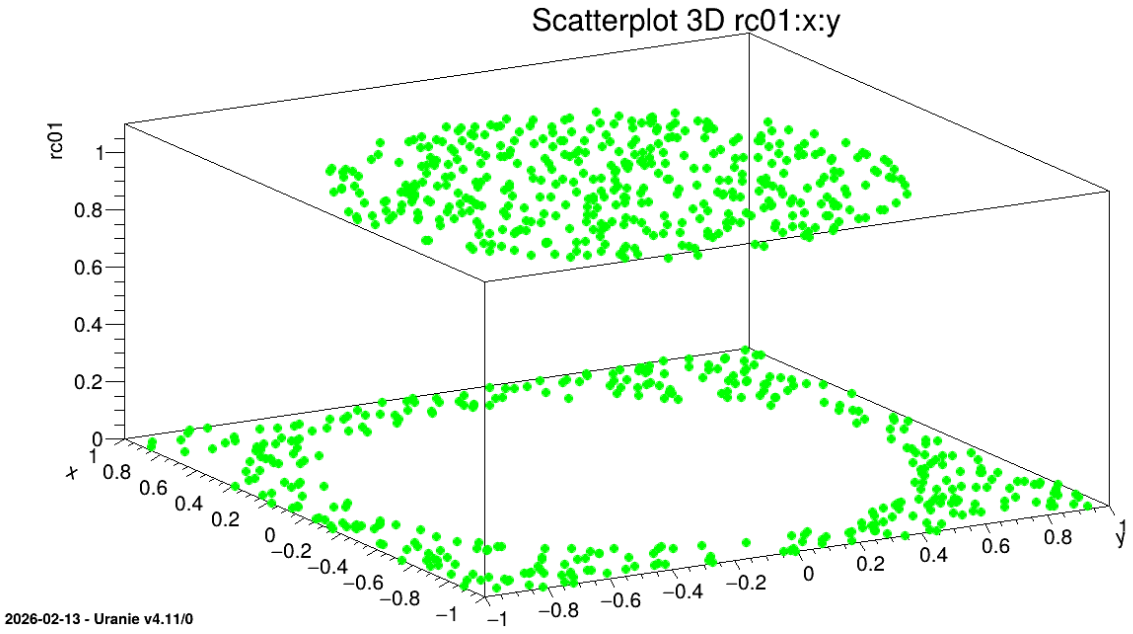
(continued from previous page)

```
*******************************
*** exportFunction lang[c++] file[uranie_ann_flowrate_loaded] name[ANNflowrate]
*** End Of exportFunction
*******************************
```

13.7.6 Macro "modelerClassificationNeuralNetworks.py"

13.7.6.1 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.

13.7.6.2 Macro Uranie

```
"""
Example of classification problem with neural network
"""
from URANIE import DataServer, Launcher, Modeler
import ROOT

nH1 = 15
nH2 = 10
nH3 = 5
```

(continues on next page)

(continued from previous page)

```
sX = "x:y"
sY = "rc01"
sYhat = sY+"_"+str(nH1)+"_"+str(nH2)+"_"+str(nH3)

# Load a database in an ASCII file
tdsu = DataServer.TDataServer("tdsu", "tds u")
tdsu.fileDataRead("problem2Classes_001000.dat")

# Split into 2 datasets for learning and testing
Niter = tdsu.getIteratorName()
tdsu.exportData("_problem2Classes_app_.dat", "", Niter+"<=800")
tdsu.exportData("_problem2Classes_val_.dat", "", Niter+">800")

tds = DataServer.TDataServer("tdsApp", "tds App for problem2Classes")
tds.fileDataRead("_problem2Classes_app_.dat")

tann = Modeler.TANNModeler(tds, sX+", %i, %i, %i,@" % (nH1, nH2, nH3)+sY)
# tann.setLog()
tann.setNormalization(Modeler.TANNModeler.kMinusOneOne)
tann.setFcnTol(1e-6)
tann.train(2, 2, "test", False)

tann.exportFunction("c++", "uranie_ann_problem2Classes", "ANNproblem2Classes")

ROOT.gROOT.LoadMacro("uranie_ann_problem2Classes.C")

tdsv = DataServer.TDataServer()
tdsv.fileDataRead("_problem2Classes_val_.dat")

# evaluate the surrogate model on the database
tlf = Launcher.TLauncherFunction(tdsv, "ANNproblem2Classes", sX, sYhat)
tlf.run()

c = ROOT.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(ROOT.kGreen)
tds.draw(sY+":"+sX)

tdsv.getTuple().SetMarkerStyle(8)
tdsv.getTuple().SetMarkerSize(0.75)
tdsv.getTuple().SetMarkerColor(ROOT.kRed)
tdsv.Draw(sYhat+":"+sX, "", "same")
```

The main `TDataServer` loads the main ASCII data file `problem2Classes_001000.dat`

```
tdsu = DataServer.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", "", Niter+"<=800")
tdsu.exportData("_problem2Classes_val_.dat", "", Niter+">800")
```

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.

```
tds = DataServer.TDataServer("tdsApp", "tds App for problem2Classes")
tds.fileDataRead("_problem2Classes_app_.dat")
tann = Modeler.TANNModeler(tds, sX+", %i, %i, %i,@" % (nH1, nH2, nH3)+sY)
tann.setNormalization(Modeler.TANNModeler.kMinusOneOne)
tann.setFcnTol(1e-6)
tann.train(2, 2, "test", False)
```

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", "ANNproblem2Classes")
```

The model is loaded from the macro `uranie_ann_problem2Classes.C` and applied on the second database with the function `ANNproblem2Classes`.

```
ROOT.gROOT.LoadMacro("uranie_ann_problem2Classes.C")
tdsv = DataServer.TDataServer()
tdsv.fileDataRead("_problem2Classes_val_.dat")
tlf = Launcher.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.

```
c = ROOT.TCanvas("c1", "Graph for the Macro modeler", 5, 64, 1270, 667)
tds.getTuple().SetMarkerStyle(8)
tds.getTuple().SetMarkerSize(1.0)
tds.getTuple().SetMarkerColor(ROOT.kGreen)
tds.draw(sY+":"+sX)

tdsv.getTuple().SetMarkerStyle(8)
tdsv.getTuple().SetMarkerSize(0.75)
tdsv.getTuple().SetMarkerColor(ROOT.kRed)
tdsv.Draw(sYhat+":"+sX, "", "same")
```

13.7.6.3 Graph

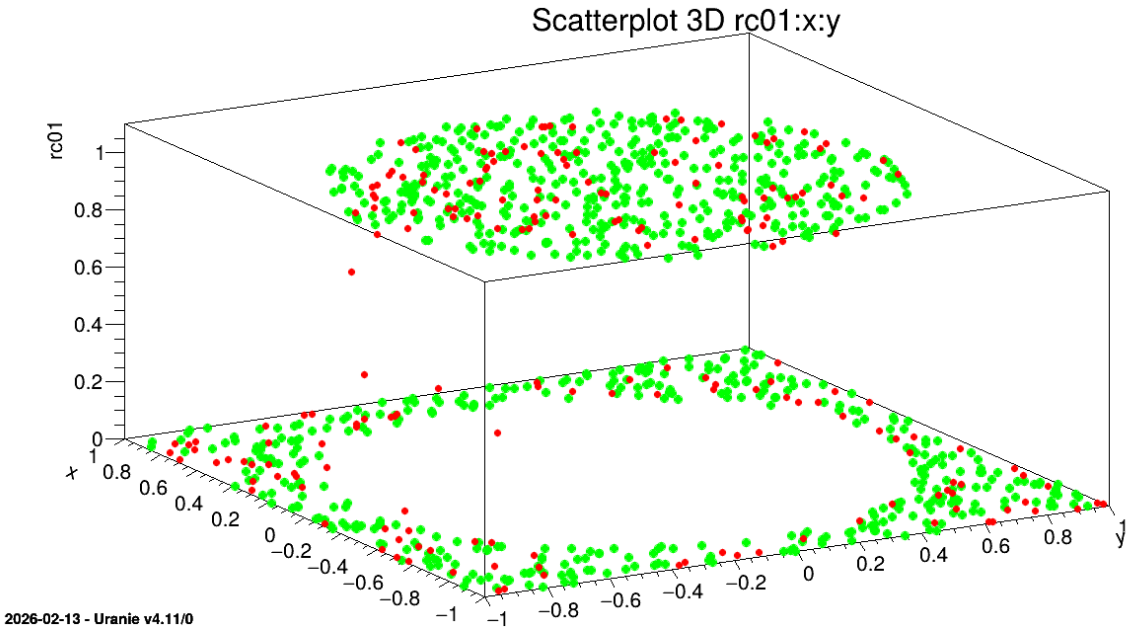


Figure 13.42: Graph of the macro **"modelerClassificationNeuralNetworks.py"**

13.7.6.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[760]
 <URANIE::WARNING> TDataServer::fileDataRead: Expected iterator tdsu__n__iter__ not␣
↪found but _tds___n__iter__ looks like an URANIE iterator => Will be used as so.
 <URANIE::WARNING> *** END of URANIE WARNING ***
 <URANIE::WARNING>
 <URANIE::WARNING>
 <URANIE::WARNING> *** URANIE WARNING ***
 <URANIE::WARNING> *** File[${SOURCEDIR}/dataSERVER/souRCE/TDataServer.cxx] Line[760]
 <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)
```

(continues on next page)

(continued from previous page)

```
 ** 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: 19 sec

******************************
*** TModeler::exportFunction lang[c++] file[uranie_ann_problem2Classes]␣
↪name[ANNproblem2Classes] soption[]

******************************

******************************
*** exportFunction lang[c++] file[uranie_ann_problem2Classes] name[ANNproblem2Classes]
*** End Of exportFunction
******************************
```

13.7.7 Macro “modelerFlowratePolynChaosRegression.py”

13.7.7.1 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 *Presentation of the problem*. The method used here is the regression one, as discussed below.

13.7.7.2 Macro Uranie

```
"""
Example of Chaos polynomial expansion using regression estimation on flowrate
"""
from URANIE import DataServer, Launcher, Modeler
import ROOT

ROOT.gROOT.LoadMacro("UserFunctions.C")

# Define the DataServer
tds = DataServer.TDataServer("tdsflowreate", "DataBase flowreate")
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))
```

(continues on next page)

(continued from previous page)

```
# Define of TNisp object
nisp = Modeler.TNisp(tds)
nisp.generateSample("QmcSobol", 500)  # State that there is a sample ...

# Compute the output variable
tlf = Launcher.TLauncherFunction(tds, "flowrateModel", "*", "ymod")
tlf.setDrawProgressBar(False)
tlf.run()

# build a chaos polynomial
pc = Modeler.TPolynomialChaos(tds, nisp)

# compute the pc coefficients using the "Regression" method
degree = 4
pc.setDegree(degree)
pc.computeChaosExpansion("Regression")

# Uncertainty and sensitivity analysis
print("Variable ymod ================")
print("Mean     = "+str(pc.getMean("ymod")))
print("Variance = "+str(pc.getVariance("ymod")))
print("First Order Indices ================")
print("Indice First Order[1] = "+str(pc.getIndexFirstOrder(0, 0)))
print("Indice First Order[2] = "+str(pc.getIndexFirstOrder(1, 0)))
print("Indice First Order[3] = "+str(pc.getIndexFirstOrder(2, 0)))
print("Indice First Order[4] = "+str(pc.getIndexFirstOrder(3, 0)))
print("Indice First Order[5] = "+str(pc.getIndexFirstOrder(4, 0)))
print("Indice First Order[6] = "+str(pc.getIndexFirstOrder(5, 0)))
print("Indice First Order[7] = "+str(pc.getIndexFirstOrder(6, 0)))
print("Indice First Order[8] = "+str(pc.getIndexFirstOrder(7, 0)))
print("Total Order Indices ================")
print("Indice Total Order[1] = "+str(pc.getIndexTotalOrder("rw", "ymod")))
print("Indice Total Order[2] = "+str(pc.getIndexTotalOrder("r", "ymod")))
print("Indice Total Order[3] = "+str(pc.getIndexTotalOrder("tu", "ymod")))
print("Indice Total Order[4] = "+str(pc.getIndexTotalOrder("tl", "ymod")))
print("Indice Total Order[5] = "+str(pc.getIndexTotalOrder("hu", "ymod")))
print("Indice Total Order[6] = "+str(pc.getIndexTotalOrder("hl", "ymod")))
print("Indice Total Order[7] = "+str(pc.getIndexTotalOrder("l", "ymod")))
print("Indice Total Order[8] = "+str(pc.getIndexTotalOrder("kw", "ymod")))

# Dump main factors up to a certain threshold
seuil = 0.99
print("Ordered functionnal ANOVA")
pc.getAnovaOrdered(seuil, 0)

print("Number of experiments = "+str(tds.getNPatterns()))

# 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
nisp = Modeler.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.

```
tlf = Launcher.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.

```
pc = Modeler.TPolynomialChaos(tds, nisp)
# compute the pc coefficients using the "Regression" method
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.

13.7.7.3 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

Variable ymod ================
Mean     = 77.65102218686935
Variance = 2079.6809330496953
First Order Indices ================
Indice First Order[1] = 0.8287315275344169
Indice First Order[2] = 1.0554825358264659e-06
Indice First Order[3] = 9.868045331678087e-08
Indice First Order[4] = 4.776680831542115e-06
Indice First Order[5] = 0.04140149326660007
Indice First Order[6] = 0.04132153620304167
Indice First Order[7] = 0.03940096615922832
Indice First Order[8] = 0.00954643619284736
Total Order Indices ================
Indice Total Order[1] = 0.8667727763800165
Indice Total Order[2] = 9.274665082296543e-06
Indice Total Order[3] = 7.017680590520732e-06
Indice Total Order[4] = 1.7194373944638416e-05
Indice Total Order[5] = 0.05419157561201169
Indice Total Order[6] = 0.05405781720242182
Indice Total Order[7] = 0.05221211558384676
Indice Total Order[8] = 0.012780873435736001
```

13.7.8 Macro “modelerFlowratePolynChaosIntegration.py”

13.7.8.1 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 *Presentation of the problem*. The method used here is the regression one, as discussed below.

13.7.8.2 Macro Uranie

```
"""
Example of Chaos polynomial expansion on flowrate
"""
from URANIE import DataServer, Launcher, Modeler
import ROOT

ROOT.gROOT.LoadMacro("UserFunctions.C")

# Define the DataServer
tds = DataServer.TDataServer("tdsflowreate", "DataBase flowreate")
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("r", 100.0, 50000.0))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))

# Define of TNisp object
nisp = Modeler.TNisp(tds)
nisp.generateSample("Petras", 5)  # State that there is a sample ...

# Compute the output variable
tlf = Launcher.TLauncherFunction(tds, "flowrateModel", "*", "ymod")
tlf.setDrawProgressBar(False)
tlf.run()

# build a chaos polynomial
pc = Modeler.TPolynomialChaos(tds, nisp)

# compute the pc coefficients using the "Integration" method
degree = 4
pc.setDegree(degree)
pc.computeChaosExpansion("Integration")

# Uncertainty and sensitivity analysis
print("Variable ymod ================")
print("Mean     = "+str(pc.getMean("ymod")))
print("Variance = "+str(pc.getVariance("ymod")))
print("First Order Indices ================")
print("Indice First Order[1] = "+str(pc.getIndexFirstOrder(0, 0)))
print("Indice First Order[2] = "+str(pc.getIndexFirstOrder(1, 0)))
print("Indice First Order[3] = "+str(pc.getIndexFirstOrder(2, 0)))
```

(continues on next page)

(continued from previous page)

```
print("Indice First Order[4] = "+str(pc.getIndexFirstOrder(3, 0)))
print("Indice First Order[5] = "+str(pc.getIndexFirstOrder(4, 0)))
print("Indice First Order[6] = "+str(pc.getIndexFirstOrder(5, 0)))
print("Indice First Order[7] = "+str(pc.getIndexFirstOrder(6, 0)))
print("Indice First Order[8] = "+str(pc.getIndexFirstOrder(7, 0)))
print("Total Order Indices ================")
print("Indice Total Order[1] = "+str(pc.getIndexTotalOrder("rw", "ymod")))
print("Indice Total Order[2] = "+str(pc.getIndexTotalOrder("r", "ymod")))
print("Indice Total Order[3] = "+str(pc.getIndexTotalOrder("tu", "ymod")))
print("Indice Total Order[4] = "+str(pc.getIndexTotalOrder("tl", "ymod")))
print("Indice Total Order[5] = "+str(pc.getIndexTotalOrder("hu", "ymod")))
print("Indice Total Order[6] = "+str(pc.getIndexTotalOrder("hl", "ymod")))
print("Indice Total Order[7] = "+str(pc.getIndexTotalOrder("l", "ymod")))
print("Indice Total Order[8] = "+str(pc.getIndexTotalOrder("kw", "ymod")))

# Dump main factors up to a certain threshold
seuil = 0.99
print("Ordered functionnal ANOVA")
pc.getAnovaOrdered(seuil, 0)

print("Number of experiments = "+str(tds.getNPatterns()))

# 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).

```
nisp = Modeler.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.

```
tlf = Launcher.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.

```
pc = Modeler.TPolynomialChaos(tds, nisp)

# compute the pc coefficients using the "Integration" method
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.

13.7.8.3 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

Variable ymod ================
Mean     = 77.65109941166783
Variance = 2078.9066967013787
First Order Indices ================
Indice First Order[1] = 0.8289221400657457
Indice First Order[2] = 1.0447660450188066e-06
Indice First Order[3] = 8.946920854938701e-12
Indice First Order[4] = 5.302986799455716e-06
Indice First Order[5] = 0.041385187477657376
Indice First Order[6] = 0.04138518747765731
Indice First Order[7] = 0.03934189598916908
Indice First Order[8] = 0.009521571788843197
Total Order Indices ================
Indice Total Order[1] = 0.8668339836271369
Indice Total Order[2] = 2.1617808194716235e-06
Indice Total Order[3] = 2.2829619916786966e-09
Indice Total Order[4] = 1.0738582524791707e-05
Indice Total Order[5] = 0.054109548638212865
Indice Total Order[6] = 0.05410954863821284
Indice Total Order[7] = 0.0520766679067344
Indice Total Order[8] = 0.012730598799757903
```

13.7.9 Macro “modelerbuildSimpleGP.py”

13.7.9.1 Objective

This macro is the one described in *Example: construction of a simple Kriging model*, 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`).

13.7.9.2 Macro Uranie

```
"""
Example of Gaussian Process building
"""
from URANIE import DataServer, Modeler

# Load observations
tdsObs = DataServer.TDataServer("tdsObs", "observations")
tdsObs.fileDataRead("utf_4D_train.dat")

# Construct the GPBuilder
gpb = Modeler.TGPBuilder(tdsObs,          # observations data
                         "x1:x2:x3:x4",   # list of input variables
                         "y",             # output variable
                         "matern3/2")     # name of the correlation function
```

(continues on next page)

(continued from previous page)

```
# Search for the optimal hyper-parameters
gpb.findOptimalParameters("ML",          # optimisation criterion
                          100,           # screening design size
                          "neldermead",  # optimisation algorithm
                          500)           # max. number of optim iterations

# Construct the kriging model
krig = gpb.buildGP()

# Display model information
krig.printLog()
```

13.7.9.3 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
******************************
```

13.7.10 Macro “modelerbuildGPInitPoint.py”

13.7.10.1 Objective

This macro is the one described in *Choice of the initial point of the optimisation*, 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.

13.7.10.2 Macro Uranie

```
"""
Example of Gaussian Process building with initial point set
"""
import numpy as np
from URANIE import DataServer, Modeler
```

(continues on next page)

(continued from previous page)

```
# Load observations
tdsObs = DataServer.TDataServer("tdsObs", "observations")
tdsObs.fileDataRead("utf_4D_train.dat")

# Construct the GPBuilder
gpb = Modeler.TGPBuilder(tdsObs, "x1:x2:x3:x4", "y", "matern3/2")

# Set the correlation function parameters
params = np.array([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 optim iterations
                          False)         # Don't reset parameters of GP builder

# Create the kriging model
krig = gpb.buildGP()

# Display model information
krig.printLog()
```

13.7.10.3 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
*******************************
```

13.7.11 Macro “modelerbuildGPWithAPriori.py”

13.7.11.1 Objective

This macro is the one described in *Deterministic trend and bayesian prior*, that creates a gaussian process with a specific trend and an *a priori* knowledge on the mean and variance of the trend parameters.

13.7.11.2 Macro Uranie

```
"""
Example of Gaussian Process building with a priori knowledge
"""
import numpy as np
from URANIE import DataServer, Modeler

# Load observations
tdsObs = DataServer.TDataServer("tdsObs", "observations")
tdsObs.fileDataRead("utf_4D_train.dat")

# Construct the GPBuilder
gpb = Modeler.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
meanPrior = np.array([0.0, 0.0, -1.0, 0.0, -0.1])
covPrior = np.array([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 optim iterations

# Construct the kriging model
krig = gpb.buildGP()

# Display model information
krig.printLog()
```

13.7.11.3 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
```

(continues on next page)

(continued from previous page)

```
 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
*******************************
```

13.7.12 Macro "modelerbuildSimpleGPEstim.py"

13.7.12.1 Objective

This macro is the one described in *Prediction of a new data set, one-by-one approach*, 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 [Bla17] for completness.

13.7.12.2 Macro Uranie

```
"""
Example of Gaussian Process building with estimation on another dataset
"""
from URANIE import DataServer, Modeler, Relauncher

# Load observations
tdsObs = DataServer.TDataServer("tdsObs", "observations")
tdsObs.fileDataRead("utf_4D_train.dat")

# Construct the GPBuilder
gpb = Modeler.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 optim iterations

# Construct the kriging model
krig = gpb.buildGP()

# Display model information
krig.printLog()

# Load the data to estimate
tdsEstim = DataServer.TDataServer("tdsEstim", "estimations")
tdsEstim.fileDataRead("utf_4D_test.dat")
```

(continues on next page)

(continued from previous page)

```
# Construction of the launcher
lanceur = Relauncher.TLauncher2(tdsEstim,          # data to estimate
                                krig,              # model used
                                "x1:x2:x3:x4",     # list of the input variables
                                "yEstim:vEstim")   # name of model's outputs

# Launch the estimations
lanceur.solverLoop()

# Display some results
tdsEstim.draw("yEstim:y")
```

13.7.12.3 Graph

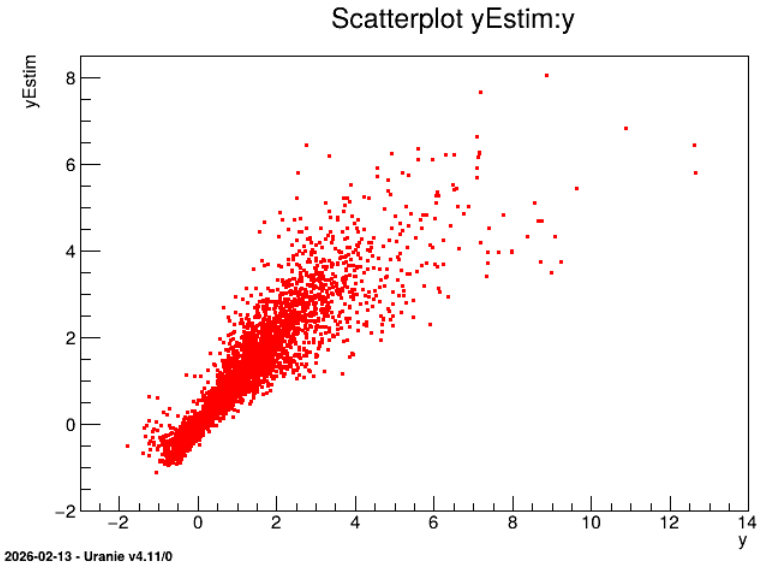


Figure 13.43: Graph of the macro **"modelerbuildSimpleGPEstim.py"**

13.7.13 Macro "modelerbuildSimpleGPEstimWithCov.py"

13.7.13.1 Objective

This macro is the one described in *Prediction of a new data set, global approach*, 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.

13.7.13.2 Macro Uranie

```
"""
Example of Gaussian Process building with estimation using full data (covariance)
"""
from URANIE import DataServer, Modeler
```

(continues on next page)

(continued from previous page)

```
import ROOT

# Load observations
tdsObs = DataServer.TDataServer("tdsObs", "observations")
tdsObs.fileDataRead("utf_4D_train.dat")

# Construct the GPBuilder
gpb = Modeler.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 optim iterations

# Construct the kriging model
krig = gpb.buildGP()

# Display model information
krig.printLog()

# Load the data to estimate
tdsEstim = DataServer.TDataServer("tdsEstim", "estimations")
tdsEstim.fileDataRead("utf_4D_test.dat")

# Reducing DB to 1000 first event (leading to cov matrix of a million value)
nS = 1000
tdsEstim.exportData("utf_4D_test_red.dat", "", "tdsEstim__n__iter__<="+str(nS))
# Reload reduced sample
tdsEstim.fileDataRead("utf_4D_test_red.dat", False, True)

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
                     "DummyOption")     # options

cTwo = None
# Produce residuals plots if true information provided
if tdsEstim.isAttribute("_Residuals_"):
    cTwo = ROOT.TCanvas("c2", "c2", 1200, 800)
    cTwo.Divide(2, 1)
    cTwo.cd(1)
    # Usual residual considering uncorrated input points
    tdsEstim.Draw("_Residuals_")
    cTwo.cd(2)
    # Corrected residuals, taking full prediction covariance matrix
    tdsEstim.Draw("_uncorrResiduals_")
```

(continues on next page)

(continued from previous page)

```
# Retrieve all the prediction covariance coefficient
tdsEstim.getTuple().SetEstimate(nS*nS)  # allocate the correct size
# Get a pointer to all values
tdsEstim.getTuple().Draw("_CovarianceMatrix_", "", "goff")
cov = tdsEstim.getTuple().GetV1()

# Put these in a matrix nicely created
Cov = ROOT.TMatrixD(nS, nS)
Cov.Use(0, nS-1, 0, nS-1, cov)

# Print it if size is reasonnable
if nS < 10:
    Cov.Print()
```

13.7.13.3 Graph

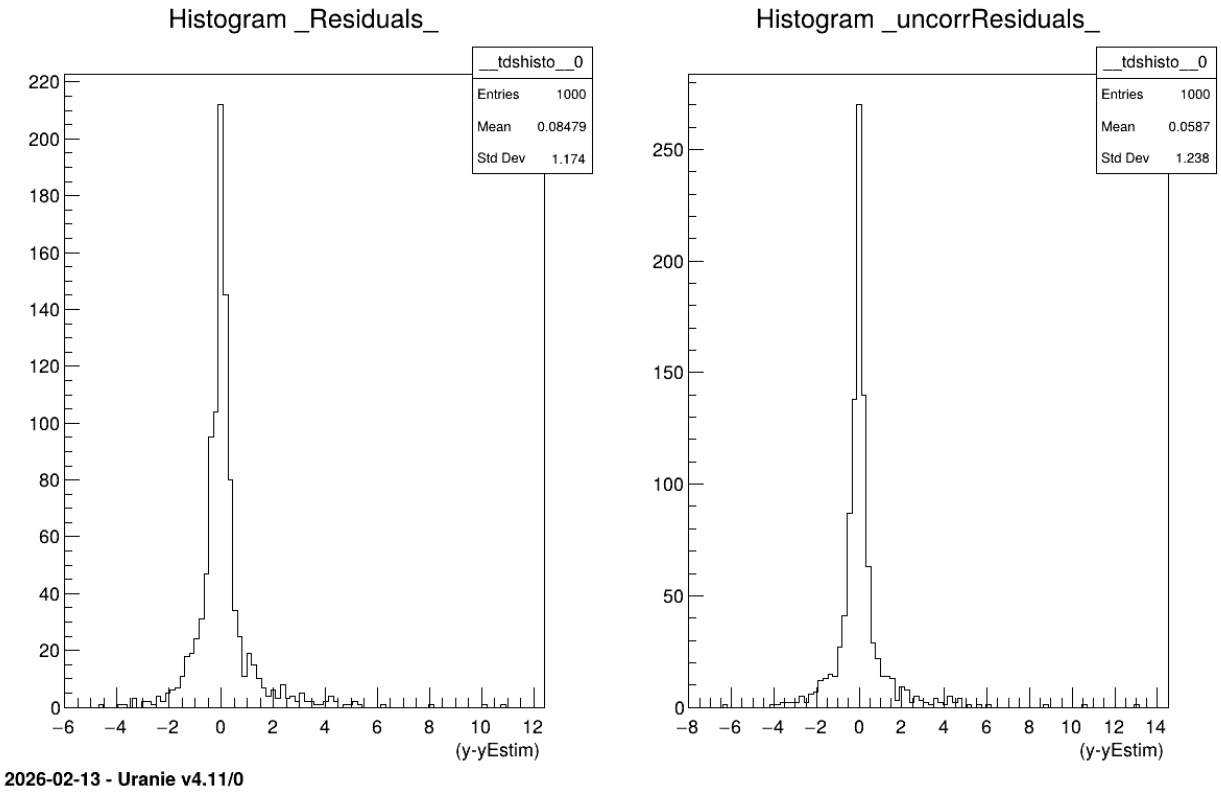


Figure 13.44: Graph of the macro **"modelerbuildSimpleGPEstimWithCov.py"**

13.7.14 Macro "modelerTestKriging.py"

13.7.14.1 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 [Bla17] 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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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.

13.7.14.2 Macro Uranie

```
"""
Example of simple GP usage and illustration of results
"""
from math import sqrt
import numpy as np
from URANIE import DataServer, Modeler, Relauncher
import ROOT

def print_solutions(tds_obs, tds_estim, title):
    """Simple function to produce the 1D plot with observation
    points, testing points along with the uncertainty band."""

    nb_obs = tds_obs.getNPatterns()
    nb_est = tds_estim.getNPatterns()
    rx_arr = np.zeros([nb_est])
    x_arr = np.zeros([nb_est])
    y_est = np.zeros([nb_est])
    y_rea = np.zeros([nb_est])
    std_var = np.zeros([nb_est])

    tds_estim.computeRank("x1")
    tds_estim.getTuple().copyBranchData(rx_arr, nb_est, "Rk_x1")
    tds_estim.getTuple().copyBranchData(x_arr, nb_est, "x1")
    tds_estim.getTuple().copyBranchData(y_est, nb_est, "yEstim")
    tds_estim.getTuple().copyBranchData(y_rea, nb_est, "y")
    tds_estim.getTuple().copyBranchData(std_var, nb_est, "vEstim")

    xarr = np.zeros([nb_est])
    yest = np.zeros([nb_est])
    yrea = np.zeros([nb_est])
    stdvar = np.zeros([nb_est])

    for i in range(nb_est):
        ind = int(rx_arr[i]) - 1
        xarr[ind] = x_arr[i]
        yest[ind] = y_est[i]
```

(continues on next page)

(continued from previous page)

```
        yrea[ind] = y_rea[i]
        stdvar[ind] = 2*sqrt(abs(std_var[i]))

    xobs = np.zeros([nb_obs])
    tds_obs.getTuple().copyBranchData(xobs, nb_obs, "x1")
    yobs = np.zeros([nb_obs])
    tds_obs.getTuple().copyBranchData(yobs, nb_obs, "y")

    gobs = ROOT.TGraph(nb_obs, xobs, yobs)
    gobs.SetMarkerColor(1)
    gobs.SetMarkerSize(1)
    gobs.SetMarkerStyle(8)
    zeros = np.zeros([nb_est])  # use doe plotting only
    gest = ROOT.TGraphErrors(nb_est, xarr, yest, zeros, stdvar)
    gest.SetLineColor(2)
    gest.SetLineWidth(1)
    gest.SetFillColor(2)
    gest.SetFillStyle(3002)
    grea = ROOT.TGraph(nb_est, xarr, yrea)
    grea.SetMarkerColor(4)
    grea.SetMarkerSize(1)
    grea.SetMarkerStyle(5)
    grea.SetTitle("Real Values")

    mgr = ROOT.TMultiGraph("toto", title)
    mgr.Add(gest, "l3")
    mgr.Add(gobs, "P")
    mgr.Add(grea, "P")
    mgr.Draw("A")
    mgr.GetXaxis().SetTitle("x_{1}")

    leg = ROOT.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()
    return [mgr, leg]

# Create dataserver and read training database
tdsObs = DataServer.TDataServer("tdsObs", "observations")
tdsObs.fileDataRead("utf-1D-train.dat")
nbObs = 6

# Canvas, divided in 2
Can = ROOT.TCanvas("can", "can", 10, 32, 1600, 700)
apad = ROOT.TPad("apad", "apad", 0, 0.03, 1, 1)
apad.Draw()
apad.Divide(2, 1)

# Name of the plot and correlation functions used
outplot = "GaussAndMatern_1D_AllPoints.png"
Correl = ["Gauss", "Matern3/2"]
```

(continues on next page)

(continued from previous page)

```
# Pointer to needed objects, created in the loop
gpb = [0, 0]
kg = [0, 0]
tdsEstim = [0, 0]
lkrig = [0, 0]

mgs = [0, 0]
legs = [0, 0]
# Looping over the two correlation function chosen
for im in range(2):

    # Create the TGPBuiler object with chosen option to find optimal parameters
    gpb[im] = Modeler.TGPBuilder(tdsObs, "x1", "y", Correl[im], "")
    gpb[im].findOptimalParameters("LOO", 100, "Subplexe", 1000)

    # Get the kriging object
    kg[im] = gpb[im].buildGP()

    # open the dataserver and read the testing basis
    tdsEstim[im] = DataServer.TDataServer("tdsEstim_"+str(im),
                                          "base de test")
    tdsEstim[im].fileDataRead("utf-1D-test.dat")

    # Applied resulting kriging on test basis
    lkrig[im] = Relauncher.TLauncher2(tdsEstim[im], kg[im], "x1",
                                      "yEstim:vEstim")
    lkrig[im].solverLoop()

    # do the plot
    apad.cd(im+1)
    mgs[im], legs[im] = print_solutions(tdsObs, tdsEstim[im], Correl[im])

    lat = ROOT.TLatex()
    lat.SetNDC()
    lat.SetTextSize(0.025)
    lat.DrawLatex(0.4, 0.61, "RMSE (by Loo) "+str(kg[im].getLooRMSE()))
    lat.DrawLatex(0.4, 0.57, "Q2 "+str(kg[im].getLooQ2()))
```

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[im] = Modeler.TGPBuilder(tdsObs, "x1", "y", Correl[im], "")
gpb[im].findOptimalParameters("LOO", 100, "Subplexe", 1000)

kg[im] = gpb[im].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[im] = Relauncher.TLauncher2(tdsEstim[im], kg[im], "x1",
                                  "yEstim:vEstim")
lkrig[im].solverLoop()
```

13.7.14.3 Graph

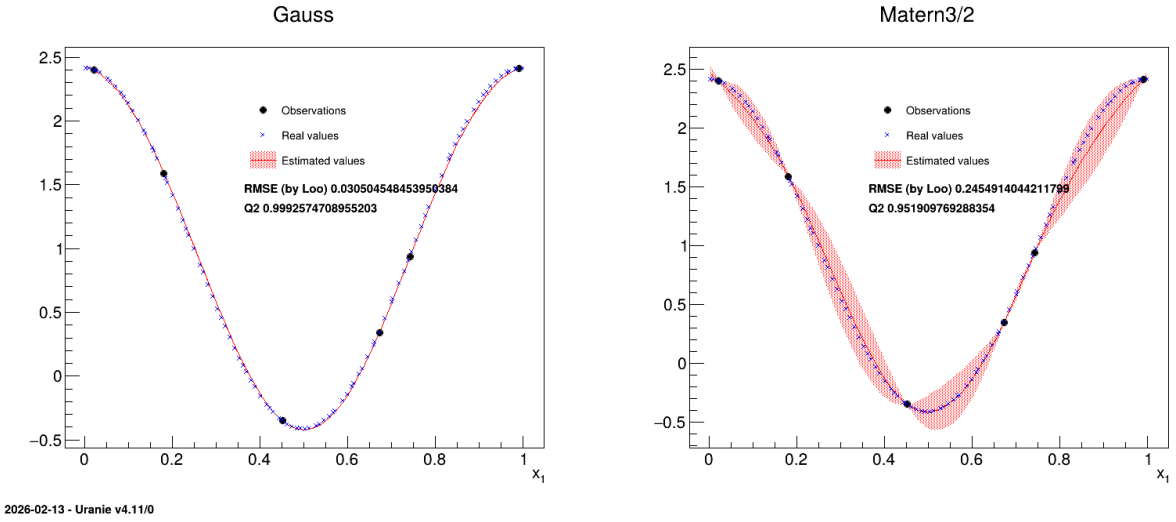


Figure 13.45: Graph of the macro **"modelerTestKriging.py"**

13.8 Macros Relauncher

The idea of this section is to show the basic usage of many of the classes defined in *The Relauncher module*, applied either on the `flowrate` functions or the `flowrate` code, whose purpose and behaviour have been already introduced in *Presentation of the problem*. All the following examples will load a tiny set of points which is gathered in the file called `flowrateUniformDesign.dat` (already introduced in *Objective*).

13.8.1 Macro "relauncherFunctionFlowrateCInt.py"

13.8.1.1 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 4.1) 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 *Important modifications going from ROOT v5 to ROOT v6* for more explanations). As this class is usually considered not thread-safe, it can only be used with a `TSequentialRun` runner.

13.8.1.2 Macro

```
"""
Example of function launching in sequential mode
"""
from URANIE import DataServer, Relauncher
import ROOT
```

(continues on next page)

(continued from previous page)

```
# Create the DataServer.TDataServer
tds = DataServer.TDataServer("foo", "test")
tds.fileDataRead("flowrateUniformDesign.dat")

# Get the attributes
rw = tds.getAttribute("rw")
r = tds.getAttribute("r")
tu = tds.getAttribute("tu")
tl = tds.getAttribute("tl")
hu = tds.getAttribute("hu")
hl = tds.getAttribute("hl")
lvar = tds.getAttribute("l")
kw = tds.getAttribute("kw")

# Create the output attribute
yhat = DataServer.TAttribute("yhat")

# Constructing the code
ROOT.gROOT.LoadMacro("UserFunctions.C")
mycode = Relauncher.TCIntEval("flowrateModel")
mycode.addInput(rw)
mycode.addInput(r)
mycode.addInput(tu)
mycode.addInput(tl)
mycode.addInput(hu)
mycode.addInput(hl)
mycode.addInput(lvar)
mycode.addInput(kw) # Adding the input attributes
mycode.addOutput(yhat)  # Adding the output attributes

# Create the sequential runner
run = Relauncher.TSequentialRun(mycode)
run.startSlave()  # Start the master (necessary even for a sequential)
if run.onMaster():

    lanceur = Relauncher.TLauncher2(tds, run)

    # resolution
    lanceur.solverLoop()
    run.stopSlave()  # Stop the slaves (necessary even for a sequential)

# Draw the result
can = ROOT.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
mycode = Relauncher.TCIntEval("flowrateModel")
mycode.addInput(rw)
mycode.addInput(r)
mycode.addInput(tu)
mycode.addInput(tl)
mycode.addInput(hu)
mycode.addInput(hl)
mycode.addInput(lvar)
mycode.addInput(kw) # Adding the input attributes
mycode.addOutput(yhat)  # Adding the output attributes
```

The methods `setInputs` and `setOutputs` are not allowed in python, so attributes has to be added one-by-one.

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
run = Relauncher.TSequentialRun(mycode)
run.startSlave()  # Start the master (necessary even for a sequential)
if run.onMaster():

    lanceur = Relauncher.TLauncher2(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.

13.8.1.3 Graph

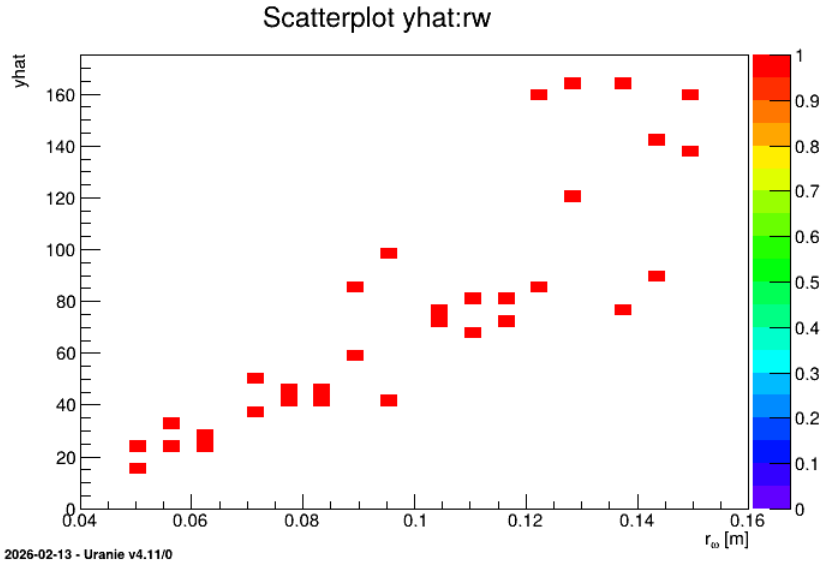


Figure 13.46: Representation of the output as a function of the first input with a colZ option

13.8.2 Macro "relauncherFunctionFlowratePython.py"

13.8.2.1 Objective

The goal of this macro is to show how to handle the Python-written function in a full python-script. This function has been presented, at least its equation (see Equation 4.1) and would be interface through the `TPythonEval` class in the Relauncher module.

13.8.2.2 Macro

```
"""
Example of python function launching
"""
import math
from URANIE import DataServer, Relauncher
import ROOT

def flowrate_model(drw, drvar, dtu, dtl, dhu, dhl, dlvar, dkw):
    """Simple flowrate model python function."""
    dnum = 2.0 * math.pi * dtu * (dhu - dhl)
    dlnronrw = math.log(drvar / drw)
    dden = dlnronrw * (1.0 + (2.0 * dlvar * dtu) / (dlnronrw * drw * drw * dkw)
                       + dtu / dtl)

    return [dnum / dden, ]


# Create the TDataServer
tds = DataServer.TDataServer("foo", "test")
tds.fileDataRead("flowrateUniformDesign.dat")

# Get the attributes
rw = tds.getAttribute("rw")
r = tds.getAttribute("r")
tu = tds.getAttribute("tu")
tl = tds.getAttribute("tl")
hu = tds.getAttribute("hu")
hl = tds.getAttribute("hl")
lvar = tds.getAttribute("l")
kw = tds.getAttribute("kw")

# Create the output attribute
yhat = DataServer.TAttribute("yhat")

# Constructing the code
mycode = Relauncher.TPythonEval(flowrate_model)
# Adding the input file
mycode.addInput(rw)
mycode.addInput(r)
mycode.addInput(tu)
mycode.addInput(tl)
mycode.addInput(hu)
mycode.addInput(hl)
mycode.addInput(lvar)
```

(continues on next page)

(continued from previous page)

```
mycode.addInput(kw)
# Adding the output file
mycode.addOutput(yhat)

# Create the sequential runner
run = Relauncher.TSequentialRun(mycode)
run.startSlave()  # Start the master (necessary even for a sequential)
if run.onMaster():

    launch = Relauncher.TLauncher2(tds, run)

    # resolution
    launch.solverLoop()
    run.stopSlave()  # Stop the slaves (necessary even for a sequential)

# Draw the result
can = ROOT.TCanvas("pouet", "foo", 1)
tds.Draw("yhat:rw", "", "colZ")
```

Obviously the code is now different from the other macros already introduced, but unless some python-specificity, the discussion has been largely done in *Macro*.

The first interesting part is the definition of the function `flowrateModel`. It is a classical python-function, for which every input is either a double, a list (for vectors) or a string. Disregarding the inner part, where the computation is done, the other interesting part is the return line: it should always be a list of all the objects that should be returned.

```
def flowrate_model(drw, drvar, dtu, dtl, dhu, dhl, dlvar, dkw):
    """Simple flowrate model python function."""
    dnum = 2.0 * math.pi * dtu * (dhu - dhl)
    dlnronrw = math.log(drvar / drw)
    dden = dlnronrw * (1.0 + (2.0 * dlvar * dtu) / (dlnronrw * drw * drw * dkw)
                       + dtu / dtl)

    return [dnum / dden, ]
```

Apart from this, the only other difference is the assessor construction which is an instance of the `TPythonEval` class as shown here:

```
# Constructing the code
mycode = Relauncher.TPythonEval(flowrate_model)
```

The macro is also leading to the creation of the following plot.

13.8.2.3 Graph

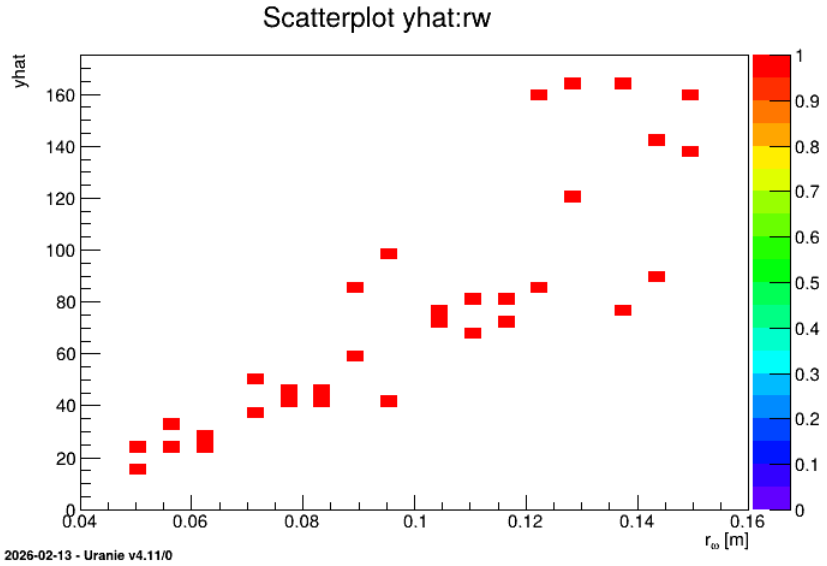


Figure 13.47: Representation of the output as a function of the first input with a colZ option

13.8.3 Macro "relauncherCodeFlowrateSequential.py"

13.8.3.1 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.

13.8.3.2 Macro

```
"""
Example of code launching in sequential mode
"""
from URANIE import DataServer, Relauncher
import ROOT

# Create the DataServer.TDataServer
tds = DataServer.TDataServer("foo", "test")
tds.fileDataRead("flowrateUniformDesign.dat")

# Get the attributes
rw = tds.getAttribute("rw")
r = tds.getAttribute("r")
tu = tds.getAttribute("tu")
tl = tds.getAttribute("tl")
hu = tds.getAttribute("hu")
hl = tds.getAttribute("hl")
lvar = tds.getAttribute("l")
kw = tds.getAttribute("kw")

# Create the output attribute
```

(continues on next page)

(continued from previous page)

```
yhat = DataServer.TAttribute("yhat")
d = DataServer.TAttribute("d")

# Set the reference input file and the key for each input attributes
fin = Relauncher.TFlatScript("flowrate_input_with_values_rows.in")
fin.addInput(rw)
fin.addInput(r)
fin.addInput(tu)
fin.addInput(tl)
fin.addInput(hu)
fin.addInput(hl)
fin.addInput(lvar)
fin.addInput(kw)

# The output file of the code
fout = Relauncher.TFlatResult("_output_flowrate_withRow_.dat")
fout.addOutput(yhat)
fout.addOutput(d)  # Passing the attributes to the output file

# Constructing the code
mycode = Relauncher.TCodeEval("flowrate -s -r")
mycode.setOldTmpDir()
mycode.addInputFile(fin)  # Adding the input file
mycode.addOutputFile(fout)  # Adding the output file

# Create the sequential runner
run = Relauncher.TSequentialRun(mycode)
run.startSlave()  # Start the master (necessary even for a sequential)
if run.onMaster():

    lanceur = Relauncher.TLauncher2(tds, run)

    # resolution
    lanceur.solverLoop()
    run.stopSlave()  # Stop the slaves (necessary even for a sequential)

# Draw the result
can = ROOT.TCanvas("pouet", "foo", 1)
tds.Draw("yhat:rw", "", "colZ")
```

Here again, a comparison is drawn with the first Relauncher macro (see *Macro*) 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
fin = Relauncher.TFlatScript("flowrate_input_with_values_rows.in")
fin.addInput(rw)
fin.addInput(r)
fin.addInput(tu)
fin.addInput(tl)
fin.addInput(hu)
```

(continues on next page)

(continued from previous page)

```
fin.addInput(hl)
fin.addInput(lvar)
fin.addInput(kw)

# The output file of the code
fout = Relauncher.TFlatResult("_output_flowrate_withRow_.dat")
fout.addOutput(yhat)
fout.addOutput(d)  # Passing the attributes to the output file

# Constructing the code
mycode = Relauncher.TCodeEval("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.

13.8.3.3 Graph

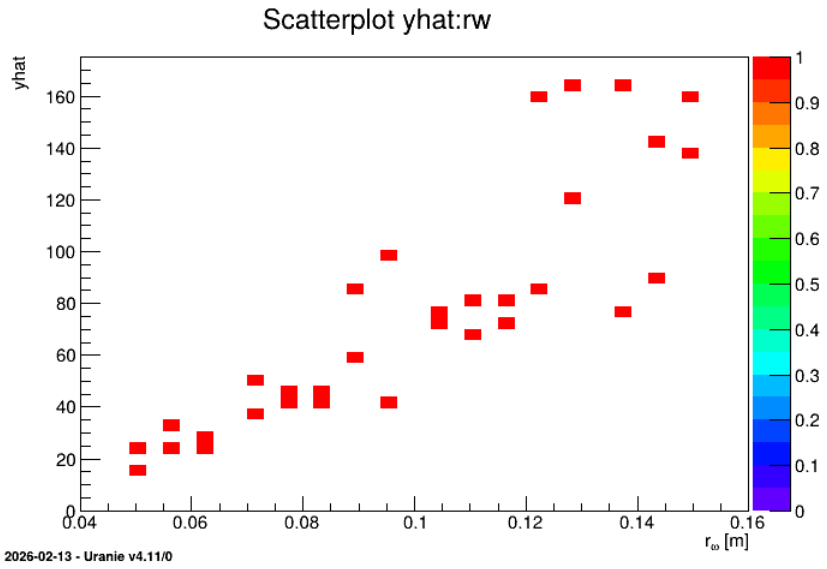


Figure 13.48: Representation of the output as a function of the first input with a colZ option

13.8.4 Macro "relauncherCodeFlowrateSequential_ConstantVar.py"

13.8.4.1 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.

13.8.4.2 Macro

```
"""
Example of code launching in sequential mode with constant variable
"""
from URANIE import DataServer, Relauncher, Sampler

# Create the TDataServer
tds = DataServer.TDataServer("foo", "test")

# Define the attribute that should be considered as constant
r = DataServer.TAttribute("r")

# Add the study attributes (min, max and nominal values)
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))

# The reference input file
sIn = "flowrate_input_with_keys.in"

nS = 15
# Generate the Design of Experiments
sampling = Sampler.TSampling(tds, "lhs", nS)
sampling.generateSample()

# Create the input files
inputFile = Relauncher.TKeyScript(sIn)
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
yhat = DataServer.TAttribute("yhat")
d = DataServer.TAttribute("d")

# Create the output files
outputFile = Relauncher.TKeyResult("_output_flowrate_withKey_.dat")
```

(continues on next page)

(continued from previous page)

```
outputFile.addOutput(yhat, "yhat")
outputFile.addOutput(d, "d")

# Create the user's evaluation function
myeval = Relauncher.TCodeEval("flowrate -s -k")
myeval.addInputFile(inputFile)
myeval.addOutputFile(outputFile)

# Create the sequential runner
run = Relauncher.TSequentialRun(myeval)
run.startSlave()  # Start the master (necessary even for a sequential)
if run.onMaster():

    lanceur = Relauncher.TLauncher2(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("*","","colsize=6")
```

Here again, a comparison is drawn with the first Relauncher macro (see *Macro*) 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
r = DataServer.TAttribute("r")

# Add the study attributes (min, max and nominal values)
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))
```

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
inputFile = Relauncher.TKeyScript(sIn)
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")
```

(continues on next page)

(continued from previous page)

```
inputFile.addInput(tds.getAttribute("hl"), "Hl")
inputFile.addInput(tds.getAttribute("l"), "L")
inputFile.addInput(tds.getAttribute("kw"), "Kw")
```

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).

```
lanceur = Relauncher.TLauncher2(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)
```

13.8.4.3 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 *
*****************************************************************************************************************
```

13.8.5 Macro “relauncherCodeFlowrateThreaded.py”

13.8.5.1 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.

13.8.5.2 Macro

```
"""
Example of code launching in threaded mode
"""
from URANIE import DataServer, Relauncher
import ROOT
# Create input attributes
rw = DataServer.TAttribute("rw")
r = DataServer.TAttribute("r")
tu = DataServer.TAttribute("tu")
tl = DataServer.TAttribute("tl")
hu = DataServer.TAttribute("hu")
hl = DataServer.TAttribute("hl")
lvar = DataServer.TAttribute("l")
kw = DataServer.TAttribute("kw")

# Create the output attribute
yhat = DataServer.TAttribute("yhat")
d = DataServer.TAttribute("d")

# Set the reference input file and the key for each input attributes
fin = Relauncher.TFlatScript("flowrate_input_with_values_rows.in")
fin.addInput(rw)
fin.addInput(r)
fin.addInput(tu)
fin.addInput(tl)
fin.addInput(hu)
fin.addInput(hl)
fin.addInput(lvar)
fin.addInput(kw)

# The output file of the code
fout = Relauncher.TFlatResult("_output_flowrate_withRow_.dat")
fout.addOutput(yhat)
fout.addOutput(d)  # Passing the attributes to the output file

# Constructing the code
mycode = Relauncher.TCodeEval("flowrate -s -r")
mycode.setOldTmpDir()
mycode.addInputFile(fin)  # Adding the input file
mycode.addOutputFile(fout)  # Adding the output file

# Fix the number of threads
nthread = 3
# Create the Threaded runner
```

(continues on next page)

(continued from previous page)

```
run = Relauncher.TThreadedRun(mycode, nthread)
run.startSlave()  # Start the master
if run.onMaster():

    # Define the DataServer
    tds = DataServer.TDataServer("tdsflowrate", "Design of Experiments")
    mycode.addAllInputs(tds)
    tds.fileDataRead("flowrateUniformDesign.dat", False, True)

    lanceur = Relauncher.TLauncher2(tds, run)

    # resolution
    lanceur.solverLoop()
    run.stopSlave()  # Stop the slaves (necessary even for a sequential)

    # Draw the result
    can = ROOT.TCanvas("pouet", "foo", 1)
    tds.Draw("yhat:rw", "", "colZ")
```

The only difference when comparing this macro to the previous one (see *Macro*) is the runner creation:

```
# Fix the number of threads
nthread = 3
# Create the Threaded runner
run = Relauncher.TThreadedRun(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.

13.8.5.3 Graph

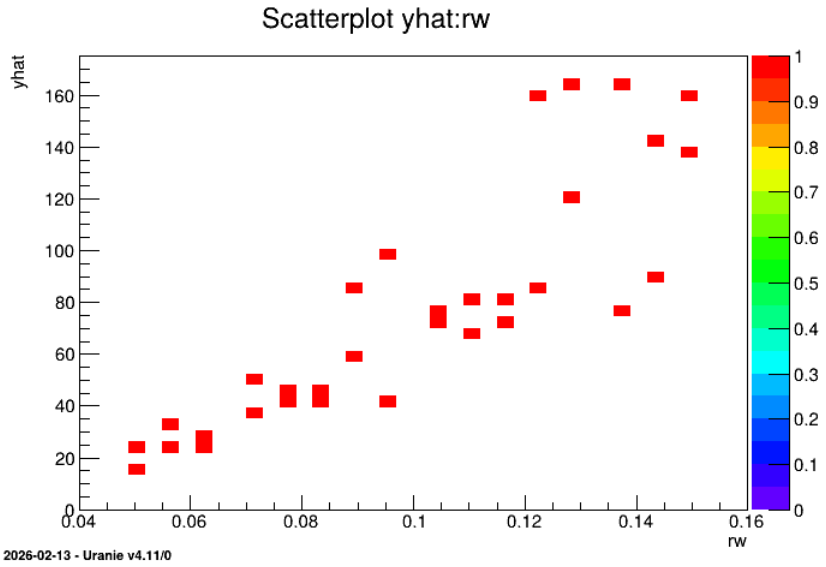


Figure 13.49: Representation of the output as a function of the first input with a colZ option

13.8.6 Macro "relauncherCodeFlowrateMPI.py"

13.8.6.1 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 *TThreadedRun*), 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.

13.8.6.2 Macro

```
"""
Example of MPi usage for code launching
"""
from URANIE import DataServer, Relauncher, MpiRelauncher
import ROOT

# Create input attributes
rw = DataServer.TAttribute("rw")
r = DataServer.TAttribute("r")
tu = DataServer.TAttribute("tu")
tl = DataServer.TAttribute("tl")
hu = DataServer.TAttribute("hu")
hl = DataServer.TAttribute("hl")
lvar = DataServer.TAttribute("l")
kw = DataServer.TAttribute("kw")

# Create the output attribute
yhat = DataServer.TAttribute("yhat")
d = DataServer.TAttribute("d")

# Set the reference input file and the key for each input attributes
fin = Relauncher.TFlatScript("flowrate_input_with_values_rows.in")
fin.addInput(rw)
fin.addInput(r)
fin.addInput(tu)
fin.addInput(tl)
fin.addInput(hu)
fin.addInput(hl)
fin.addInput(lvar)
fin.addInput(kw)

# The output file of the code
fout = Relauncher.TFlatResult("_output_flowrate_withRow_.dat")
fout.addOutput(yhat)
fout.addOutput(d)  # Passing the attributes to the output file

# Instanciation de mon code
mycode = Relauncher.TCodeEval("flowrate -s -r")
# mycode.setOldTmpDir()
mycode.addInputFile(fin)
mycode.addOutputFile(fout)
```

(continues on next page)

(continued from previous page)

```
# Create the MPI runner
run = MpiRelauncher.TMpiRun(mycode)
run.startSlave()
if run.onMaster():

    # Define the DataServer
    tds = DataServer.TDataServer("tdsflowrate",
                                 "Design of Experiments for Flowrate")
    mycode.addAllInputs(tds)
    tds.fileDataRead("flowrateUniformDesign.dat", False, True)

    lanceur = Relauncher.TLauncher2(tds, run)

    # resolution
    lanceur.solverLoop()

    tds.exportData("_output_testFlowrateMPI_py_.dat")

    ROOT.SetOwnership(run, True)
    run.stopSlave()
```

Here the first difference when comparing this macro to the previous one (see *Macro*) is the runner creation:

```
# Create the MPI runner
run = MpiRelauncher.TMpiRun(mycode)
```

The `TThreadedRun` object becomes a `TMpiRun` object whose construction only requests a pointer to the assessor.

Another line is different as it is specific to the language: because of the way ROOT and python deal with object destruction (though the garbage collector approach for the latter), there is problem in the way one of the main key method for MPI treatment `MPI_Finalize` is called. To prevent this from happening in python the following line should be added as soon as the runner object is created:

```
ROOT.SetOwnership(run, True)
```

It allows ROOT to destroy the object injected, calling the finalize method in order for every slave to be properly released. 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:

```
python relauncherCodeFlowrateMPI.py
```

Instead, the command line should start with the `mpirun` command as such:

```
mpirun -np N python relauncherCodeFlowrateMPI.py
```

where the *N* part should be replaced by the number of requested threads. Once run, this macro also leads to the following plots.

Beware never to use the `-i` argument with the python command line as the macro would never end.

13.8.6.3 Graph

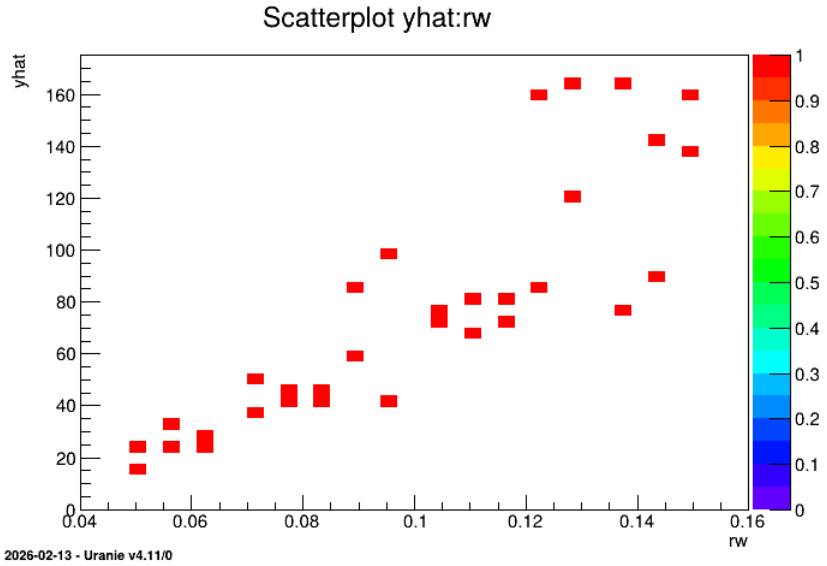


Figure 13.50: Representation of the output as a function of the first input with a colZ option

13.8.7 Macro "relauncherCodeFlowrateSequentialFailure.py"

13.8.7.1 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.

13.8.7.2 Macro

```
"""
Example of code launching in sequential mode with failure
"""
from URANIE import DataServer, Relauncher
import ROOT
# Create the DataServer.TDataServer
tds = DataServer.TDataServer("foo", "test")
tds.fileDataRead("flowrateUniformDesign.dat")

# Get the attributes
rw = tds.getAttribute("rw")
r = tds.getAttribute("r")
tu = tds.getAttribute("tu")
tl = tds.getAttribute("tl")
hu = tds.getAttribute("hu")
hl = tds.getAttribute("hl")
lvar = tds.getAttribute("l")
kw = tds.getAttribute("kw")
```

(continues on next page)

(continued from previous page)

```
# Create the output attribute
yhat = DataServer.TAttribute("yhat")
d = DataServer.TAttribute("d")

# Set the reference input file and the key for each input attributes
fin = Relauncher.TFlatScript("flowrate_input_with_values_rows.in")
fin.addInput(rw)
fin.addInput(r)
fin.addInput(tu)
fin.addInput(tl)
fin.addInput(hu)
fin.addInput(hl)
fin.addInput(lvar)
fin.addInput(kw)

# The output file of the code
fout = Relauncher.TFlatResult("_output_flowrate_withRow_.dat")
fout.addOutput(yhat)  # Passing the attributes to the output file
fout.addOutput(d)  # Passing the attributes to the output file

# Constructing the code
mycode = Relauncher.TCodeEval("flowrate -s -rf")
mycode.addInputFile(fin)  # Adding the input file
mycode.addOutputFile(fout)  # Adding the output file

# Create the sequential runner
run = Relauncher.TSequentialRun(mycode)
run.startSlave()  # Start the master (necessary even for a sequential)

if run.onMaster():

    lanceur = Relauncher.TLauncher2(tds, run)

    # Store the wrong calculation
    error = DataServer.TDataServer("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
can = ROOT.TCanvas("pouet", "foo", 1)
tds.Draw("hu:hl")
```

Here there are very few differences with the one already introduced in *Macro*. The first one is obviously the command line which is called using `"-rf"` argument, the f being introduced for failure.

```
mycode = Relauncher.TCodeEval("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
error = DataServer.TDataServer("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 13.51) 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 *Console*

```
# dump all wrong configurations
error.getTuple().SetScanField(-1)
error.scan("*")
```

The first part of the console output shown in *Console* 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_IkSFzC ; 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_IkSFzC/_output_flowrate_withRow_.dat
```

This pattern is repeated every time a configuration is wrong.

13.8.7.3 Graph

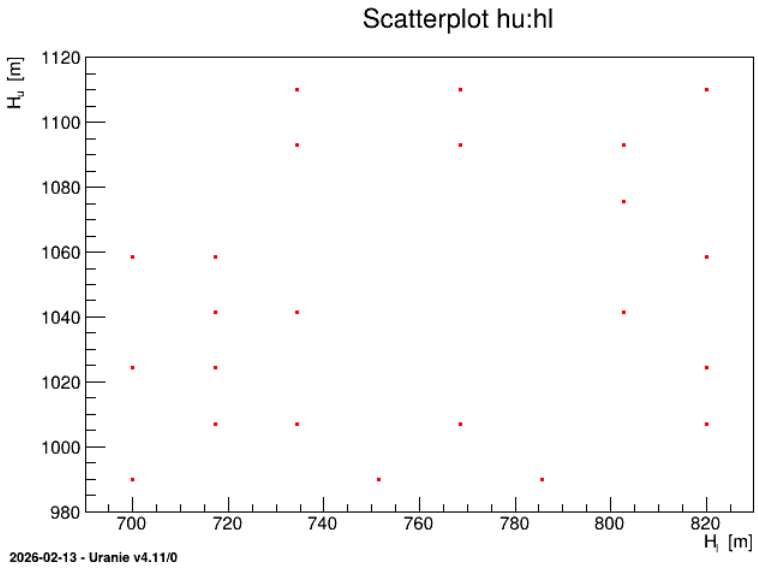


Figure 13.51: Representation of the output data point when the code is asked to fail on purpose.

13.8.7.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                      Copyright (C) 2013-2026 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Thu Feb 12, 2026

Command cd ${RUNNINGDIR}/URA_IkSFzC ; 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_IkSFzC/_output_flowrate_withRow_.dat
Command cd ${RUNNINGDIR}/URA_IkSFzC ; 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_IkSFzC/_output_flowrate_withRow_.dat
Command cd ${RUNNINGDIR}/URA_IkSFzC ; 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_IkSFzC/_output_flowrate_withRow_.dat
Command cd ${RUNNINGDIR}/URA_IkSFzC ; 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_IkSFzC/_output_flowrate_withRow_.dat
Command cd ${RUNNINGDIR}/URA_IkSFzC ; 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_IkSFzC/_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␣
```

(continues on next page)

(continued from previous page)

```
↪*     83.77 *
*        4 *        23 *    0.1233 *  16733.33 *     63070 *      63.1 *   1041.43 *    785.71 *      1680 *     12045␣
↪*     86.73 *
**********************************************************************************************************************************************
```

13.8.8 Macro "relauncherCodeMultiTypeKey.py"

13.8.8.1 Objective

The objective of this macro is to test the case where vectors and strings are produced as outputs, using the code described in *Producing outputs*, 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
```

13.8.8.2 Macro Uranie

```
"""
Example of multitype code launching
"""
from URANIE import DataServer, Relauncher, Sampler
import ROOT

# Create the DataServer.TDataServer and create the seed attribute
tds = DataServer.TDataServer("foo", "multitype usecase")
tds.addAttribute(DataServer.TUniformDistribution("seed", 0, 100000))

# Create DOE
tsam = Sampler.TSampling(tds, "lhs", 100)
tsam.generateSample()

# Create output attribute pointers
```

(continues on next page)

(continued from previous page)

```
w1 = DataServer.TAttribute("w1", DataServer.TAttribute.kString)
w2 = DataServer.TAttribute("w2", DataServer.TAttribute.kString)
v1 = DataServer.TAttribute("v1", DataServer.TAttribute.kVector)
v2 = DataServer.TAttribute("v2", DataServer.TAttribute.kVector)
f1 = DataServer.TAttribute("f1")

# Create the input files
inputFile = Relauncher.TFlatScript("multitype_input.dat")
inputFile.addInput(tds.getAttribute("seed"))

# Create the output files
outputFile = Relauncher.TKeyResult("_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
myeval = Relauncher.TCodeEval("multitype -mtKey")
myeval.addInputFile(inputFile)  # Add the input file
myeval.addOutputFile(outputFile)  # Add the output file

# Create the runner
runner = Relauncher.TSequentialRun(myeval)

# Start the slaves
runner.startSlave()
if runner.onMaster():

    # Create the launcher
    lanceur = Relauncher.TLauncher2(tds, runner)
    lanceur.solverLoop()

    # Stop the slave processes
    runner.stopSlave()

# Produce control plot
Can = ROOT.TCanvas("Can", "Can", 10, 10, 1000, 800)
pad = ROOT.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
w1 = DataServer.TAttribute("w1", DataServer.TAttribute.kString)
w2 = DataServer.TAttribute("w2", DataServer.TAttribute.kString)
v1 = DataServer.TAttribute("v1", DataServer.TAttribute.kVector)
```

(continues on next page)

(continued from previous page)

```
v2 = DataServer.TAttribute("v2", DataServer.TAttribute.kVector)
f1 = DataServer.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 *Macro "relauncherCodeFlowrateSequential.py"*) with the only difference being that the output file is a key type one. It results in the following plots.

13.8.8.3 Graph

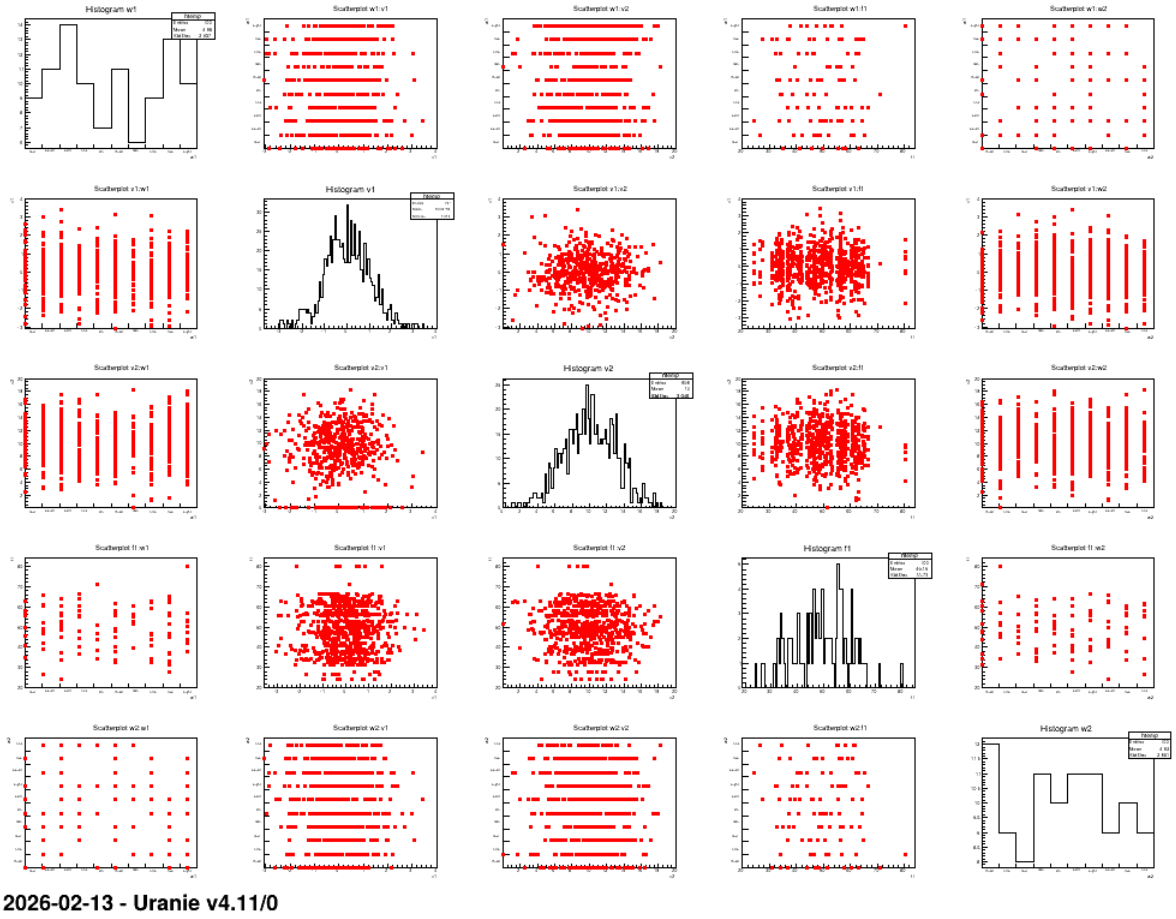


Figure 13.52: Graph of the macro **"relauncherCodeMultiTypeKey.py"**

13.8.9 Macro "relauncherCodeMultiTypeKeyEmptyVectors.py"

13.8.9.1 Objective

The objective of this macro is to test the case where vectors and strings are produced as outputs, using the code described in *Producing outputs*, with a Key format, obtained by doing:

```
multitype -mtKey -empty
```

Unlike what's done to in *Macro "relauncherCodeMultiTypeKey.py"*, 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,
↪15.6614,10.3452,9.41101,7.47887 ]
f1 = 32.2723
w2 = eight
```

13.8.9.2 Macro Uranie

```
"""
Example of code launching for which empty vectors are possible
"""
from URANIE import DataServer, Relauncher, Sampler
import ROOT
# Create the DataServer.TDataServer and create the seed attribute
tds = DataServer.TDataServer("foo", "multitype usecase")
tds.addAttribute(DataServer.TUniformDistribution("seed", 0, 100000))

# Create DOE
tsam = Sampler.TSampling(tds, "lhs", 100)
tsam.generateSample()

# Create output attribute pointers
w1 = DataServer.TAttribute("w1", DataServer.TAttribute.kString)
w2 = DataServer.TAttribute("w2", DataServer.TAttribute.kString)
v1 = DataServer.TAttribute("v1", DataServer.TAttribute.kVector)
v2 = DataServer.TAttribute("v2", DataServer.TAttribute.kVector)
f1 = DataServer.TAttribute("f1")

# Create the input files
inputFile = Relauncher.TFlatScript("multitype_input.dat")
inputFile.addInput(tds.getAttribute("seed"))

# Create the output files
outputFile = Relauncher.TKeyResult("_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
myeval = Relauncher.TCodeEval("multitype -mtKey -empty")
myeval.addInputFile(inputFile)  # Add the input file
myeval.addOutputFile(outputFile)  # Add the output file

# Create the runner
runner = Relauncher.TSequentialRun(myeval)

# Start the slaves
runner.startSlave()
```

(continues on next page)

(continued from previous page)

```
if runner.onMaster():

    # Create the launcher
    lanceur = Relauncher.TLauncher2(tds, runner)
    lanceur.solverLoop()

    # Stop the slave processes
    runner.stopSlave()

# Produce control plot
Can = ROOT.TCanvas("Can", "Can", 10, 10, 1000, 800)
pad = ROOT.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 *Macro Uranie*. 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
outputFile = Relauncher.TKeyResult("_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 13.53.

```
# Produce control plot
Can = ROOT.TCanvas("Can", "Can", 10, 10, 1000, 800)
pad = ROOT.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 *Macro "relauncherCodeMultiTypeKeyEmptyVectorsAsFailure.py"*.

13.8.9.3 Graph

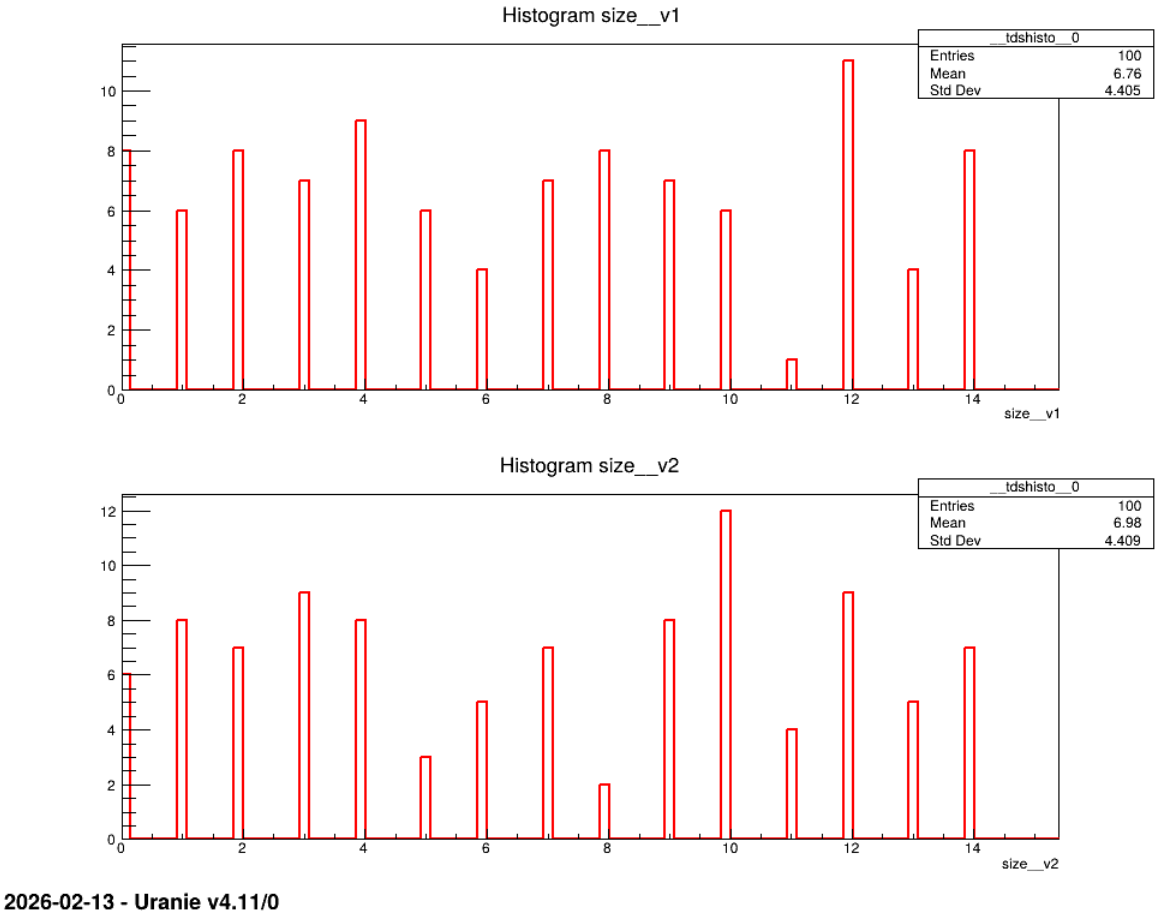


Figure 13.53: Graph of the macro **"relauncherCodeMultiTypeKeyEmptyVectors.py"**

13.8.10 Macro "relauncherCodeMultiTypeKeyEmptyVectorsAsFailure.py"

13.8.10.1 Objective

The objective of this macro is to test the case where vectors and strings are produced as outputs, using the code described in *Producing outputs*, with a Key format, obtained by doing:

```
multitype -mtKey -empty
```

Unlike what's done to in *Macro "relauncherCodeMultiTypeKey.py"*, 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
```

(continues on next page)

(continued from previous page)

```
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
```

13.8.10.2 Macro Uranie

```
"""
Example of code launching with empy vectors as failure
"""
from URANIE import DataServer, Relauncher, Sampler
import ROOT

# Create the DataServer.TDataServer and create the seed attribute
tds = DataServer.TDataServer("foo", "multitype usecase")
tds.addAttribute(DataServer.TUniformDistribution("seed", 0, 100000))

# Create DOE
tsam = Sampler.TSampling(tds, "lhs", 100)
tsam.generateSample()

# Create output attribute pointers
w1 = DataServer.TAttribute("w1", DataServer.TAttribute.kString)
w2 = DataServer.TAttribute("w2", DataServer.TAttribute.kString)
v1 = DataServer.TAttribute("v1", DataServer.TAttribute.kVector)
v2 = DataServer.TAttribute("v2", DataServer.TAttribute.kVector)
f1 = DataServer.TAttribute("f1")

# Create the input files
inputFile = Relauncher.TFlatScript("multitype_input.dat")
inputFile.addInput(tds.getAttribute("seed"))

# Create the output files
outputFile = Relauncher.TKeyResult("_output_multitype_mt_Key_.dat")
outputFile.addOutput(w1, "w1")
outputFile.addOutput(v1, "v1")
```

(continues on next page)

(continued from previous page)

```
outputFile.addOutput(v2, "v2")
outputFile.addOutput(f1, "f1")
outputFile.addOutput(w2, "w2")

# Create the user's evaluation function
myeval = Relauncher.TCodeEval("multitype -mtKey -empty")
myeval.addInputFile(inputFile)  # Add the input file
myeval.addOutputFile(outputFile)  # Add the output file

# Create the runner
runner = Relauncher.TSequentialRun(myeval)

# Start the slaves
runner.startSlave()
if runner.onMaster():

    # Create the launcher
    lanceur = Relauncher.TLauncher2(tds, runner)

    # Store the wrong calculation
    error = DataServer.TDataServer("WrongComputations", "pouet")
    lanceur.setSaveError(error)

    lanceur.solverLoop()

    # dump all wrong configurations
    error.getTuple().SetScanField(-1)
    error.scan("*")

    # Stop the slave processes
    runner.stopSlave()

# Produce control plot
Can = ROOT.TCanvas("Can", "Can", 10, 10, 1000, 800)
pad = ROOT.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 *Macro Uranie*. 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
error = DataServer.TDataServer("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
```

(continues on next page)

(continued from previous page)

```
error.getTuple().SetScanField(-1)
error.scan("*")
```

The final part is the way to represent the results: as for the use-case macro discussed in *Macro "relauncherCodeMultiTypeKey.py"*, all data are plotted in a pair plot and this is summarised in Figure 13.54. 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 *Console*. This time (unlike the failure in *Macro "relauncherCodeFlowrateSequentialFailure.py"*) 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.

13.8.10.3 Graph

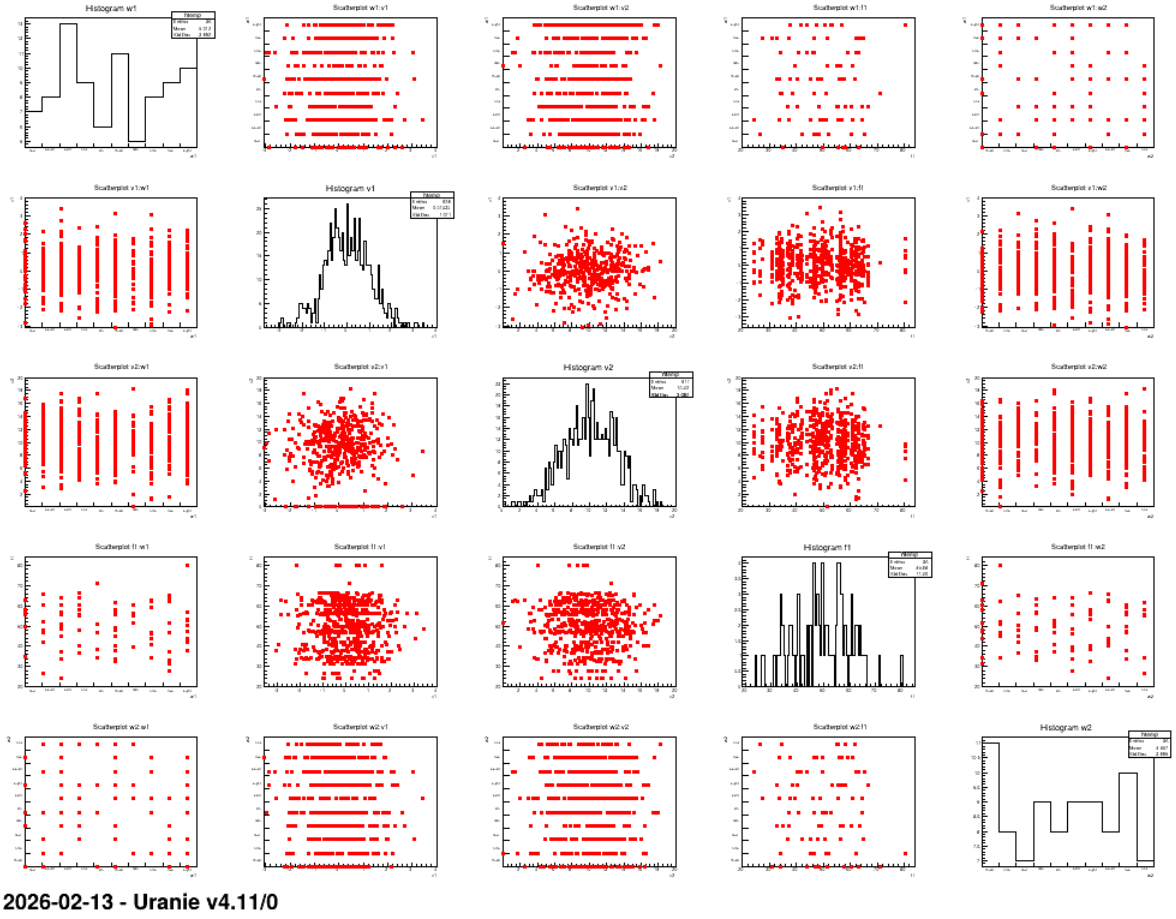


Figure 13.54: Graph of the macro **"relauncherCodeMultiTypeKeyEmptyVectorsAsFailure.py"**

13.8.10.4 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026
```

(continues on next page)

(continued from previous page)

```
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
**********************************
*    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 *
**********************************
```

13.8.11 Macro “relauncherCodeReadMultiType.py”

13.8.11.1 Objective

The objective of this macro is to test the case where vectors and strings are used as inputs, using the code described in *Reading inputs*, 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
```

(continues on next page)

(continued from previous page)

```
-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.
↪146879916125e-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.
↪066403357312e+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,
```

(continues on next page)

(continued from previous page)

```
↪15.6614,10.3452,9.41101,7.47887 ]
f1 = 32.2723
w2 = eight
```

The resulting output file, named `_output_multitype_readmt_Key_.dat` looks like:

```
thev1 = -0.2397650482929
thev2 = 9.321006265935
```

13.8.11.2 Macro Uranie

```
"""
Example of multitype code launching in sequential mode
"""
from URANIE import DataServer, Relauncher
import ROOT

# inputs
tds = DataServer.TDataServer("foo", "TDS for flowrate")
tds.fileDataRead("readmultitype_sampling.dat")

# Input attribute
w1 = tds.getAttribute("w1")
w2 = tds.getAttribute("w2")
v1 = tds.getAttribute("v1")
v2 = tds.getAttribute("v2")
f1 = tds.getAttribute("f1")

# output attribute
thev1 = DataServer.TAttribute("thev1")
thev2 = DataServer.TAttribute("thev2")
ROOT.gSystem.Exec("multitype -mtKey")
# Create the output files
inputFile = Relauncher.TKeyScript("_output_multitype_mt_Key_condensate_.dat")
inputFile.addInput(w1, "w1")
inputFile.addInput(v1, "v1")
inputFile.addInput(v2, "v2")
inputFile.addInput(f1, "f1")
inputFile.addInput(w2, "w2")

# Create the output files
outputFile = Relauncher.TKeyResult("_output_multitype_readmt_Key_.dat")
outputFile.addOutput(thev1, "thev1")
outputFile.addOutput(thev2, "thev2")

# Create the user's evaluation function
myeval = Relauncher.TCodeEval("multitype -ReadmtKey")
myeval.addInputFile(inputFile)
myeval.addOutputFile(outputFile)

runner = Relauncher.TSequentialRun(myeval)
runner.startSlave()
```

(continues on next page)

(continued from previous page)

```
if runner.onMaster():

    # Create the launcher
    lanceur = Relauncher.TLauncher2(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:

```
ROOT.gSystem.Exec("multitype -mtKey")
```

The rest is very common and a screenshot of the result displayed in console is provided in the following subsection.

13.8.11.3 Console

```
************************************
*    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 *
************************************
```

13.8.12 Macro “relauncherComposeMultitypeAndReadMultiType.py”

13.8.12.1 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 *Producing outputs*) and use these vectors and strings as inputs, using the code described in *Reading inputs*.

13.8.12.2 Macro Uranie

```
"""
Example of code composition with multitype code
"""
from URANIE import DataServer, Relauncher
import ROOT
```

(continues on next page)

(continued from previous page)

```
# inputs
tds = DataServer.TDataServer("foo", "TDS for flowrate")
tds.fileDataRead("multitype_sampling.dat")

# output attributes...
# ... for code 1
w1 = DataServer.TAttribute("w1", DataServer.TAttribute.kString)
w2 = DataServer.TAttribute("w2", DataServer.TAttribute.kString)
v1 = DataServer.TAttribute("v1", DataServer.TAttribute.kVector)
v2 = DataServer.TAttribute("v2", DataServer.TAttribute.kVector)
f1 = DataServer.TAttribute("f1")

# ... for code 2
thev1 = DataServer.TAttribute("thev1")
thev2 = DataServer.TAttribute("thev2")

# ============================================================
# ========================== Code 1 ==========================
# ============================================================

# Create the input files
inputFile1 = Relauncher.TFlatScript("multitype_input.dat")
inputFile1.addInput(tds.getAttribute("seed"))

ROOT.gSystem.Exec("multitype -mtKey")
# Create the output files
outputFile1 = Relauncher.TKeyResult("_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
eval1 = Relauncher.TCodeEval("multitype -mtKey")
eval1.addInputFile(inputFile1)
eval1.addOutputFile(outputFile1)

# ============================================================
# ========================== Code 2 ==========================
# ============================================================

# Create the output files
inputFile2 = Relauncher.TKeyScript("_output_multitype_mt_Key_condensate_.dat")
inputFile2.addInput(w1, "w1")
inputFile2.addInput(v1, "v1")
inputFile2.addInput(v2, "v2")
inputFile2.addInput(f1, "f1")
inputFile2.addInput(w2, "w2")

# Create the output files
```

(continues on next page)

(continued from previous page)

```
outputFile2 = Relauncher.TKeyResult("_output_multitype_readmt_Key_.dat")
outputFile2.addOutput(thev1, "thev1")
outputFile2.addOutput(thev2, "thev2")

# Create the user's evaluation function
eval2 = Relauncher.TCodeEval("multitype -ReadmtKey")
eval2.addInputFile(inputFile2)
eval2.addOutputFile(outputFile2)

# ================================================================
# ====================== Composition =============================
# ================================================================

# Create the composition
compo = Relauncher.TComposeEval()
# Add the code one-by-one, in the right order
compo.addEval(eval1)
compo.addEval(eval2)

# Create the runner by providing the TComposeEval
runner = Relauncher.TSequentialRun(compo)

runner.startSlave()  # compulsory
if runner.onMaster():  # compulsory

    # Create the launcher
    lanceur = Relauncher.TLauncher2(tds, runner)
    lanceur.solverLoop()  # run the code

    # Stop the slave processes
    runner.stopSlave()

    # Get the results
    tds.exportData("pouet_py.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 *Macro Uranie*.

```
w1 = DataServer.TAttribute("w1", DataServer.TAttribute.kString)
w2 = DataServer.TAttribute("w2", DataServer.TAttribute.kString)
v1 = DataServer.TAttribute("v1", DataServer.TAttribute.kVector)
v2 = DataServer.TAttribute("v2", DataServer.TAttribute.kVector)
f1 = DataServer.TAttribute("f1")
```

The output attributes are defined for the first code, as in in *Macro Uranie*.

```
thev1 = DataServer.TAttribute("thev1")
thev2 = DataServer.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:

```
compo = Relauncher.TComposeEval()
# Add the code one-by-one, in the right order
compo.addEval(eval1)
compo.addEval(eval2)
```

The rest is very common and a screenshot of the result displayed in console is provided in the following subsection.

13.8.12.3 Console

```
************************************
*    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 *
************************************
```

13.8.13 Macro “relauncherCodeFlowrateSequential_TemporaryVar.py”

13.8.13.1 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.

13.8.13.2 Macro

```
"""
Example of code launching in sequential mode with temporary variable
"""
from URANIE import DataServer, Relauncher, Sampler

def increase_d(local_x):
    """Dummy function that increase by one the input."""
    local_y = local_x + 1
    return [local_y, ]


# Create the TDataServer
tds = DataServer.TDataServer("foo", "test")

# Define the attribute that should be considered as constant
r = DataServer.TAttribute("r")
```

(continues on next page)

(continued from previous page)

```
# Add the study attributes (min, max and nominal values)
tds.addAttribute(DataServer.TUniformDistribution("rw", 0.05, 0.15))
tds.addAttribute(DataServer.TUniformDistribution("tu", 63070.0, 115600.0))
tds.addAttribute(DataServer.TUniformDistribution("tl", 63.1, 116.0))
tds.addAttribute(DataServer.TUniformDistribution("hu", 990.0, 1110.0))
tds.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 820.0))
tds.addAttribute(DataServer.TUniformDistribution("l", 1120.0, 1680.0))
tds.addAttribute(DataServer.TUniformDistribution("kw", 9855.0, 12045.0))

# The reference input file
sIn = "flowrate_input_with_keys.in"

nS = 15
# Generate the Design of Experiments
sampling = Sampler.TSampling(tds, "lhs", nS)
sampling.generateSample()

# Create the input files
inputFile = Relauncher.TKeyScript(sIn)
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
yhat = DataServer.TAttribute("yhat")
d = DataServer.TAttribute("d")

# Create the output files
outputFile = Relauncher.TKeyResult("_output_flowrate_withKey_.dat")
outputFile.addOutput(yhat, "yhat")
outputFile.addOutput(d, "d")

# Create the user's evaluation function
eval1 = Relauncher.TCodeEval("flowrate -s -k")
eval1.addInputFile(inputFile)
eval1.addOutputFile(outputFile)

# Create a second evaluation function that uses d to change it slightly
incd = DataServer.TAttribute("incd")
eval2 = Relauncher.TPythonEval(increase_d)
eval2.addInput(d)
eval2.addOutput(incd)

# Create the composition
evalC = Relauncher.TComposeEval()
# Add the code one-by-one, in the right order
evalC.addEval(eval1)
```

(continues on next page)

(continued from previous page)

```
evalC.addEval(eval2)

# Create the sequential runner
run = Relauncher.TSequentialRun(evalC)
run.startSlave()  # Start the master (necessary even for a sequential)
if run.onMaster():

    lanceur = Relauncher.TLauncher2(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 useful 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("*", "", "colsize=6")
```

Here again, a comparison is drawn with the macro in which we set an attribute to a constant value (see *Macro "relauncher-CodeFlowrateSequential_ConstantVar.py"*), 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.

```
def increase_d(local_x):
    """Dummy function that increase by one the input."""
    local_y = local_x + 1
    return [local_y, ]
```

The rest is exactly as for *Macro "relauncherCodeFlowrateSequential_ConstantVar.py"*, up to the interface with the newy create function:

```
# Create a second evaluation function that uses d to change it slightly
incd = DataServer.TAttribute("incd")
eval2 = Relauncher.TPythonEval(increase_d)
eval2.addInput(d)
eval2.addOutput(incd)

# Create the composition
evalC = Relauncher.TComposeEval()
# Add the code one-by-one, in the right order
evalC.addEval(eval1)
evalC.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 *Console* for consistency check.

```
lanceur = Relauncher.TLauncher2(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 useful for another evaluator
lanceur.addTemporary(d)
lanceur.addConstantValue(r, 108)
```

13.8.13.3 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 *
*****************************************************************************************************
```

13.9 Macros Reoptimizer

13.9.1 Macro “reoptimizeHollowBarCode.py”

13.9.1.1 Objective

The objective of the macro is to optimize the section of the hollow bar defined in *Sizing of a hollow bar example problem* using the NLopt solvers (reducing it to a single-criterion optimisation as already explained in *Local solver*. 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).

13.9.1.2 Macro Uranie

```
"""
Example of hollow bar optimisation
"""
import numpy as np
from URANIE import DataServer, Relauncher, Reoptimizer
import ROOT

# variables
x = DataServer.TAttribute("x", 0.0, 1.0)
y = DataServer.TAttribute("y", 0.0, 1.0)
thick = DataServer.TAttribute("thick")  # thickness
sect = DataServer.TAttribute("sect")  # section of the pipe
dist = DataServer.TAttribute("dist")  # distortion

# Creating the TCodeEval, dumping output of the dummy python in output file
python_exec = "python3"
if ROOT.gSystem.GetBuildArch() == "win64":
    python_exec = python_exec[:-1]
code = Relauncher.TCodeEval(python_exec + " bar.py > bartoto.dat")

# Pass the python script itself as input. inputs are modified in bar.py
inputfile = Relauncher.TKeyScript("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
outputfile = Relauncher.TFlatResult("bartoto.dat")
outputfile.addOutput(thick)
outputfile.addOutput(sect)
outputfile.addOutput(dist)
code.addOutputFile(outputfile)

# Create a runner
runner = Relauncher.TSequentialRun(code)
runner.startSlave()  # Usual Relauncher construction

if runner.onMaster():

    # Create the TDS
```

(continues on next page)

(continued from previous page)

```
    tds = DataServer.TDataServer("vizirDemo", "Param de l'opt vizir")
    tds.addAttribute(x)
    tds.addAttribute(y)

    # Choose a solver
    solv = Reoptimizer.TNloptCobyla()
    # solv = Reoptimizer.TNloptBobyqa()
    # solv = Reoptimizer.TNloptPraxis()
    # solv = Reoptimizer.TNloptNelderMead()
    # solv = Reoptimizer.TNloptSubplexe()

    # Create the single-objective constrained optimizer
    opt = Reoptimizer.TNlopt(tds, runner, solv)

    # add the objective
    opt.addObjective(sect)  # minimizing the section

    # and the constrains
    constrDist = Reoptimizer.TLesserFit(14)
    opt.addConstraint(dist, constrDist)  # distortion (dist < 14)
    positiv = Reoptimizer.TGreaterFit(0.4)
    opt.addConstraint(thick, positiv)  # thickness (thick > 0.4)

    # Starting point and maximum evaluation
    point = np.array([0.9, 0.2])
    opt.setStartingPoint(len(point), point)
    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
x = DataServer.TAttribute("x", 0.0, 1.0)
y = DataServer.TAttribute("y", 0.0, 1.0)
thick = DataServer.TAttribute("thick")  # thickness
sect = DataServer.TAttribute("sect")  # section of the pipe
dist = DataServer.TAttribute("dist")  # distortion
```

where the first two are the input ones while the last ones are computed using the provided code (as explained in *Sizing of a hollow bar example problem*). This code is configured through these lines:

```
# Creating the TCodeEval, dumping output of the dummy python in output file
code = Relauncher.TCodeEval(python_exec + " bar.py > bartoto.dat")

# Pass the python script itself as input. inputs are modified in bar.py
```

(continues on next page)

(continued from previous page)

```
inputfile = Relauncher.TKeyScript("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
outputfile = Relauncher.TFlatResult("bartoto.dat")
outputfile.addOutput(thick)
outputfile.addOutput(sect)
outputfile.addOutput(dist)
code.addOutputFile(outputfile)

# Create a runner
runner = Relauncher.TSequentialRun(code)
runner.startSlave()  # Usual Relauncher construction
```

The usual Relauncher construction is followed, using a `TSequentialRun` runner and the solver is chosen in these lines:

```
# Choose a solver
solv = Reoptimizer.TNloptCobyla()
# solv = Reoptimizer.TNloptBobyqa()
# solv = Reoptimizer.TNloptPraxis()
# solv = Reoptimizer.TNloptNelderMead()
# solv = Reoptimizer.TNloptSubplexe()
```

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
opt = Reoptimizer.TNlopt(tds, runner, solv)

# add the objective
opt.addObjective(sect)  # minimizing the section

# and the constrains
constrDist = Reoptimizer.TLesserFit(14)
opt.addConstraint(dist, constrDist)  # distortion (dist < 14)
positiv = Reoptimizer.TGreaterFit(0.4)
opt.addConstraint(thick, positiv)  # 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
point = np.array([0.9, 0.2])
opt.setStartingPoint(len(point), point)
opt.setMaximumEval(1000)

opt.solverLoop()  # running the optimization
```

13.9.1.3 Console

This macro leads to the following result

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

|....:....|....:....|....:....|....:....
|*************************************************************************
*    Row   * vizirDemo *       x.x *       y.y * thick * sect * dist.dist *
*************************************************************************
*        0 *         0 * 0.5173156 * 0.1173173 * 0.399 * 0.25 * 13.999986 *
*************************************************************************
```

13.9.2 Macro “reoptimizeHollowBarCodeMultiStart.py”

13.9.2.1 Objective

The objective of the macro is to optimize the section of the hollow bar defined in *Sizing of a hollow bar example problem* using the NLopt solvers (reducing it to a single-criterion optimisation as already explained in *Local solver*. It is largely based on the previous macro, the main change being the fact that we allow different starting points.

13.9.2.2 Macro Uranie

```
"""
Example of hollow bar optimisation in multistart mode
"""
import numpy as np
from URANIE import DataServer, Relauncher, Reoptimizer
import ROOT

# variables
x = DataServer.TAttribute("x", 0.0, 1.0)
y = DataServer.TAttribute("y", 0.0, 1.0)
thick = DataServer.TAttribute("thick")  # thickness
sect = DataServer.TAttribute("sect")  # section of the pipe
dist = DataServer.TAttribute("dist")  # distortion

# Creating the TCodeEval, dumping output of the dummy python in an output file
python_exec = "python3"
if ROOT.gSystem.GetBuildArch() == "win64":
    python_exec = python_exec[:-1]
code = Relauncher.TCodeEval(python_exec + " bar.py > bartoto.dat")

# Pass the python script itself as input. inputs are modified in bar.py
inputfile = Relauncher.TKeyScript("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
outputfile = Relauncher.TFlatResult("bartoto.dat")
```

(continues on next page)

(continued from previous page)

```
outputfile.addOutput(thick)
outputfile.addOutput(sect)
outputfile.addOutput(dist)
code.addOutputFile(outputfile)

# Create a runner
runner = Relauncher.TSequentialRun(code)
runner.startSlave()  # Usual Relauncher construction

if runner.onMaster():

    # Create the TDS
    tds = DataServer.TDataServer("vizirDemo", "Param de l'opt vizir")
    tds.addAttribute(x)
    tds.addAttribute(y)

    # Choose a solver
    solv = Reoptimizer.TNloptCobyla()
    # solv = Reoptimizer.TNloptBobyqa()
    # solv = Reoptimizer.TNloptPraxis()
    # solv = Reoptimizer.TNloptNelderMead()
    # solv = Reoptimizer.TNloptSubplexe()

    # Create the single-objective constrained optimizer
    opt = Reoptimizer.TNlopt(tds, runner, solv)

    # add the objective
    opt.addObjective(sect)  # minimizing the section

    # and the constrains
    constrDist = Reoptimizer.TLesserFit(14)
    opt.addConstraint(dist, constrDist)  # distortion (dist < 14)
    positiv = Reoptimizer.TGreaterFit(0.4)
    opt.addConstraint(thick, positiv)  # thickness (thick > 0.4)

    # Starting point
    p1 = np.array([0.9, 0.2])
    p2 = np.array([0.7, 0.1])
    p3 = np.array([0.5, 0.4])
    opt.setStartingPoint(len(p1), p1)
    opt.setStartingPoint(len(p2), p2)
    opt.setStartingPoint(len(p3), p3)

    # Set maximum evaluation
    opt.setMaximumEval(1000)

    opt.solverLoop()  # running the optimization

    # Stop the slave processes
    runner.stopSlave()

# solution
```

(continues on next page)

(continued from previous page)

```
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 *Macro "reoptimizeHollowBarCode.py"*. The only difference is highlighted here:

```
# Starting point
p1 = np.array([0.9, 0.2])
p2 = np.array([0.7, 0.1])
p3 = np.array([0.5, 0.4])
opt.setStartingPoint(len(p1), p1)
opt.setStartingPoint(len(p2), p2)
opt.setStartingPoint(len(p3), p3)
```

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

```
runner = Relauncher.TSequentialRun(code)
```

to, for instance in our case with 3 starting points

```
runner=Relauncher.TThreadedRun(code,4)
```

13.9.2.3 Console

This macro leads to the following result

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

|....:....|....:....|....:....|....:....|....:....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 *
*************************************************************************
```

13.9.3 Macro "reoptimizeHollowBarCodevizir.py"

13.9.3.1 Objective

The objective of the macro is to optimize the section and distortion of the hollow bar defined in *Sizing of a hollow bar example problem* using the evolutionary solvers. This can be done with different solvers, the one chosen here being the `TVizirGenetic` one.

13.9.3.2 Macro Uranie

```
"""
Example of hollow bar optimisation with Vizir
"""
from URANIE import DataServer, Relauncher, Reoptimizer
import ROOT

# variables
x = DataServer.TAttribute("x", 0.0, 1.0)
y = DataServer.TAttribute("y", 0.0, 1.0)
thick = DataServer.TAttribute("thick")  # thickness
sect = DataServer.TAttribute("sect")  # section of the pipe
dist = DataServer.TAttribute("dist")  # distortion

# Creating TCodeEval, dumping output in an output
python_exec = "python3"
if ROOT.gSystem.GetBuildArch() == "win64":
    python_exec = python_exec[:-1]
code = Relauncher.TCodeEval(python_exec + " bar.py > bartoto.dat")

# Pass the python script as a input. Inputs are modified in bar.py
inputfile = Relauncher.TKeyScript("bar.py")
inputfile.addInput(x, "x")
inputfile.addInput(y, "y")
code.addInputFile(inputfile)

# precise the output file in which to read the three outputs
outputfile = Relauncher.TFlatResult("bartoto.dat")
outputfile.addOutput(thick)
outputfile.addOutput(sect)
outputfile.addOutput(dist)
code.addOutputFile(outputfile)

# Create a runner
runner = Relauncher.TSequentialRun(code)
runner.startSlave()  # Usual Relauncher construction

if runner.onMaster():

    # Create the TDS
    tds = DataServer.TDataServer("vizirDemo", "Param de l'opt vizir")
    tds.addAttribute(x)
    tds.addAttribute(y)

    # create the vizir genetic solver
    solv = Reoptimizer.TVizirGenetic()
    # Size of the population and maximum number of evaluation
    solv.setSize(200, 15000)

    # Create the multi-objective constrained optimizer
    opt = Reoptimizer.TVizir2(tds, runner, solv)
```

(continues on next page)

(continued from previous page)

```
    # add the objective
    opt.addObjective(sect)  # minimizing the section
    opt.addObjective(dist)  # minimizing the distortion

    # and the constrains
    positiv = Reoptimizer.TGreaterFit(0.4)
    opt.addConstraint(thick, positiv)  # on thickness (thick > 0.4)

    opt.solverLoop()  # running the optimization

    # Stop the slave processes
    runner.stopSlave()

fig1 = ROOT.TCanvas("fig1", "Pareto Zone", 5, 64, 1270, 667)
phi = 12
theta = 30
pad1 = ROOT.TPad("pad1", "", 0, 0.03, 1, 1)
pad2 = ROOT.TPad("pad2", "", 0, 0.03, 1, 1)
pad2.SetFillStyle(4000)  # will be transparent

pad1.Draw()
pad1.Divide(2, 1)
pad1.cd(1)
ROOT.gPad.SetPhi(phi)
ROOT.gPad.SetTheta(theta)

ROOT.gStyle.SetLabelSize(0.03)
ROOT.gStyle.SetLabelSize(0.03, "Y")
ROOT.gStyle.SetLabelSize(0.03, "Z")

tds.getTuple().Draw("sect:y:x")
# Get the TH3 to change Z axis color
htemp = ROOT.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)
ROOT.gPad.SetFillStyle(4000)
ROOT.gPad.SetPhi(phi)
ROOT.gPad.SetTheta(theta)
tds.getTuple().SetMarkerColor(4)
tds.getTuple().Draw("dist:y:x")
htemp = ROOT.gPad.GetPrimitive("htemp")
htemp.SetTitle("")
htemp.GetZaxis().SetLabelColor(4)
htemp.GetZaxis().SetAxisColor(4)
htemp.GetZaxis().SetTitleColor(4)
```

(continues on next page)

(continued from previous page)

```
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
x = DataServer.TAttribute("x", 0.0, 1.0)
y = DataServer.TAttribute("y", 0.0, 1.0)
thick = DataServer.TAttribute("thick")  # thickness
sect = DataServer.TAttribute("sect")  # section of the pipe
dist = DataServer.TAttribute("dist")  # distortion
```

where the first two are the input ones while the last ones are computed using the provided code (as explained in *Sizing of a hollow bar example problem*). This code is configured through these lines

```
# Creating TCodeEval, dumping output in an output
code = Relauncher.TCodeEval(python_exec + " bar.py > bartoto.dat")

# Pass the python script as a input. Inputs are modified in bar.py
inputfile = Relauncher.TKeyScript("bar.py")
inputfile.addInput(x, "x")
inputfile.addInput(y, "y")
code.addInputFile(inputfile)

# precise the output file in which to read the three outputs
outputfile = Relauncher.TFlatResult("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
solv = Reoptimizer.TVizirGenetic()
# Size of the population and maximum number of evaluation
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
opt = Reoptimizer.TVizir2(tds, runner, solv)

# add the objective
opt.addObjective(sect)  # minimizing the section
```

(continues on next page)

(continued from previous page)

```
opt.addObjective(dist)   # minimizing the distortion

# and the constrains
positiv = Reoptimizer.TGreaterFit(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.

13.9.3.3 Graph

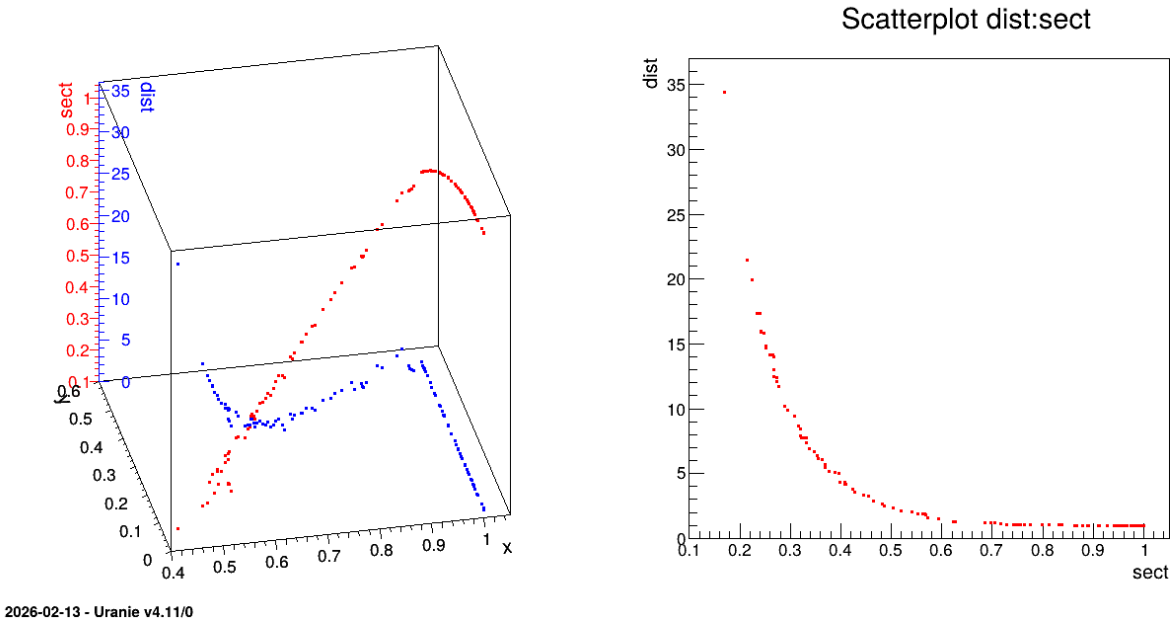


Figure 13.55: Graph of the macro **"reoptimizeHollowBarCodeVizir.py"**

13.9.4 Macro "reoptimizeHollowBarVizirMoead.py"

13.9.4.1 Objective

The objective of the macro is to optimize the section and distortion of the hollow bar defined in *Sizing of a hollow bar example problem* 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 [Bla17].

13.9.4.2 Macro Uranie

```
"""
Example of hollow bar optimisation with Moead approach
"""
from array import array
from URANIE import DataServer, Relauncher, Reoptimizer
import ROOT
```

(continues on next page)

(continued from previous page)

```
# variables
x = DataServer.TAttribute("x", 0.0, 1.0)
y = DataServer.TAttribute("y", 0.0, 1.0)
thick = DataServer.TAttribute("thick")  # thickness
sect = DataServer.TAttribute("sect")  # section of the pipe
dist = DataServer.TAttribute("dist")  # distortion

ROOT.gROOT.LoadMacro("UserFunctions.C")

# Creating the assessor using the analytical function
code = Relauncher.TCIntEval("barAllCost")
code.addInput(x)
code.addInput(y)
code.addOutput(thick)
code.addOutput(sect)
code.addOutput(dist)

# Create a runner
runner = Relauncher.TSequentialRun(code)
runner.startSlave()  # Usual Relauncher construction

NB = 20
nMax = 3000
total = 4*NB
if runner.onMaster():
    # ================================================
    # ========= Classical Vizir implementation =========
    # ================================================

    # Create the TDS
    tds_viz = DataServer.TDataServer("vizirDemo", "Vizir parameter dataser")
    tds_viz.addAttribute(x)
    tds_viz.addAttribute(y)

    # create the vizir genetic solver
    solv_viz = Reoptimizer.TVizirGenetic()
    # Size of the population, and a maximum number of evaluation
    solv_viz.setSize(NB, nMax)

    # Create the multi-objective constrained optimizer
    opt_viz = Reoptimizer.TVizir2(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
    positiv = Reoptimizer.TGreaterFit(0.4)
    opt_viz.addConstraint(thick, positiv)  # on thickness (thick > 0.4)

    opt_viz.solverLoop()  # running the optimization

    # ================================================
```

(continues on next page)

(continued from previous page)

```
    # ============== MOEAD implementation ==============
    # ==================================================

    # Create the TDS
    tds_moead = DataServer.TDataServer("vizirDemo", "Vizir parameter dataser")
    tds_moead.addAttribute(x)
    tds_moead.addAttribute(y)

    # create the vizir genetic solver
    solv_moead = Reoptimizer.TVizirGenetic()
    solv_moead.setMoeadDiversity(NB)
    solv_moead.setStoppingCriteria(1)
    solv_moead.setSize(0, nMax)

    # Create the multi-objective constrained optimizer
    opt_moead = Reoptimizer.TVizir2(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
    # Preparing canvas
    fig1 = ROOT.TCanvas("fig1", "Pareto Zone", 5, 64, 1270, 667)
    pad1 = ROOT.TPad("pad1", "", 0, 0.03, 1, 1)
    pad1.Draw()
    pad1.Divide(2, 1)
    pad1.cd(1)

    # extracting data to construct graphs
    viz = array('d', [0. for i in range(total)])
    # There is always one more point in moead
    moe = array('d', [0. for i in range(total+4)])
    tds_viz.getTuple().extractData(viz, total, "x:y:sect:dist", "", "column")
    tds_moead.getTuple().extractData(moe, total+4,
                                     "x:y:sect:dist", "", "column")

    set_viz = ROOT.TGraph(NB, array('d', [viz[i] for i in range(NB)]),
                          array('d', [viz[NB+i] for i in range(NB)]))
    fr_viz = ROOT.TGraph(NB, array('d', [viz[2*NB+i] for i in range(NB)]),
                         array('d', [viz[3*NB+i] for i in range(NB)]))
    set_viz.SetMarkerColor(4)
    set_viz.SetMarkerStyle(20)
    set_viz.SetMarkerSize(0.8)
    fr_viz.SetMarkerColor(4)
```

(continues on next page)

(continued from previous page)

```
    fr_viz.SetMarkerStyle(20)
    fr_viz.SetMarkerSize(0.8)

    set_moe = ROOT.TGraph(NB+1, array('d', [moe[i] for i in range(NB+1)]),
                          array('d', [moe[NB+1+i] for i in range(NB+1)]))
    fr_moe = ROOT.TGraph(NB, array('d', [moe[2*(NB+1)+i] for i in range(NB+1)]),
                         array('d', [moe[3*(NB+1)+i] for i in range(NB+1)]))
    set_moe.SetMarkerColor(2)
    set_moe.SetMarkerStyle(20)
    set_moe.SetMarkerSize(0.8)
    fr_moe.SetMarkerColor(2)
    fr_moe.SetMarkerStyle(20)
    fr_moe.SetMarkerSize(0.8)

    # Legend
    ROOT.gStyle.SetLegendBorderSize(0)
    leg = ROOT.TLegend(0.25, 0.75, 0.55, 0.89)
    leg.AddEntry(set_viz, "Vizir algo", "p")
    leg.AddEntry(set_moe, "MOEAD algo", "p")

    # Pareto Set
    set_mg = ROOT.TMultiGraph()
    set_mg.Add(set_viz)
    set_mg.Add(set_moe)
    set_mg.Draw("aP")
    set_mg.SetTitle("Pareto Set")
    set_mg.GetXaxis().SetTitle("x")
    set_mg.GetYaxis().SetTitle("y")
    leg.Draw()
    ROOT.gPad.Update()

    # Pareto Front
    pad1.cd(2)
    fr_mg = ROOT.TMultiGraph()
    fr_mg.Add(fr_viz)
    fr_mg.Add(fr_moe)
    fr_mg.Draw("aP")
    fr_mg.SetTitle("Pareto front")
    fr_mg.GetXaxis().SetTitle("Section")
    fr_mg.GetYaxis().SetTitle("Distortion")
    leg.Draw()
    ROOT.gPad.Update()
```

The variables are defined as follow:

```
x = DataServer.TAttribute("x", 0.0, 1.0)
y = DataServer.TAttribute("y", 0.0, 1.0)
thick = DataServer.TAttribute("thick")  # thickness
sect = DataServer.TAttribute("sect")  # section of the pipe
dist = DataServer.TAttribute("dist")  # distortion
```

where the first two are the input ones while the last ones are computed using the provided code (as explained in *Sizing of a hollow bar example problem*). This code is configured through these lines

```
# Creating the assessor using the analytical function
code = Relauncher.TCIntEval("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

```
# create the vizir genetic solver
solv_viz = Reoptimizer.TVizirGenetic()
# Size of the population, and a maximum number of evaluation
solv_viz.setSize(NB, 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
opt_viz = Reoptimizer.TVizir2(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
positiv = Reoptimizer.TGreaterFit(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
solv_moead = Reoptimizer.TVizirGenetic()
solv_moead.setMoeadDiversity(NB)
solv_moead.setStoppingCriteria(1)
solv_moead.setSize(0, nMax)
```

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.

13.9.4.3 Graph

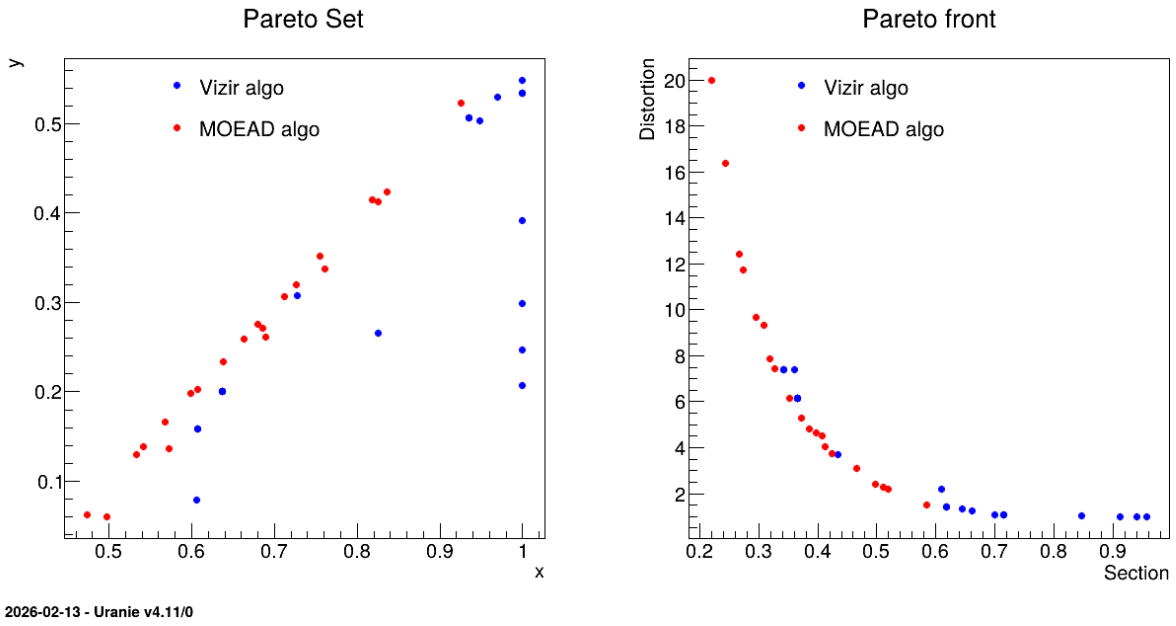


Figure 13.56: Graph of the macro **"reoptimizeHollowBarVizirMoead.py"**

13.9.5 Macro "reoptimizeHollowBarVizirSplitRuns.py"

13.9.5.1 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 *Sizing of a hollow bar example problem* is used to obtain a nice Pareto set/front.

13.9.5.2 Macro Uranie

```
"""
Example of hollow bar optimisation in split runs
"""
from URANIE import DataServer, Relauncher, Reoptimizer
import ROOT

TOLERANCE = 0.001
NBmaxEVAL = 1200
SIZE = 500

def launch_vizir(run_number, fig_one):
    """Factorised function to run the Vizir analaysis."""
    # variables
    xvar = DataServer.TAttribute("x", 0.0, 1.0)
    yvar = DataServer.TAttribute("y", 0.0, 1.0)
```

(continues on next page)

(continued from previous page)

```
    thick = DataServer.TAttribute("thick")
    sect = DataServer.TAttribute("sect")
    defor = DataServer.TAttribute("defor")

    code = Relauncher.TCIntEval("barAllCost")
    code.addInput(xvar)
    code.addInput(yvar)
    code.addOutput(thick)
    code.addOutput(sect)
    code.addOutput(defor)

    # Create a runner
    runner = Relauncher.TSequentialRun(code)
    runner.startSlave()

    # Output to state whether convergence is reached
    has_converged = False
    if runner.onMaster():

        # Create the TDS
        tds = DataServer.TDataServer("vizirDemo", "Param de l'opt vizir")
        tds.addAttribute(xvar)
        tds.addAttribute(yvar)

        solv = Reoptimizer.TVizirGenetic()
        # Name of the file that will contain
        filename = "genetic.dump"
        # whether = Test if a 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, filename):
            print("Restarting Vizir")
        else:
            solv.setSize(SIZE, NBmaxEVAL)

        # Create the multi-objective constrained optimizer
        opt = Reoptimizer.TVizir2(tds, runner, solv)
        opt.setTolerance(TOLERANCE)
        # add the objective
        opt.addObjective(sect)
        opt.addObjective(defor)
        positiv = Reoptimizer.TGreaterFit(0.4)
        opt.addConstraint(thick, positiv)

        # resolution
        opt.solverLoop()
        has_converged = opt.isConverged()
        # Stop the slave processes
        runner.stopSlave()

        fig_one.cd(run_number+1)
        tds.getTuple().SetMarkerColor(2)
```

(continues on next page)

(continued from previous page)

```
        tds.draw("defor:sect")
        tit = "Run number "+str(run_number+1)
        if has_converged:
            tit += ": Converged !"
        ROOT.gPad.GetPrimitive("__tdshisto__0").SetTitle(tit)

    return has_converged


ROOT.gROOT.LoadMacro("UserFunctions.C")
# Delete previous file if it exists
ROOT.gSystem.Unlink("genetic.dump")

finished = False
i = 0
fig1 = ROOT.TCanvas("fig1", "fig1", 1200, 800)
fig1.Divide(2, 2)
while not finished:
    finished = launch_vizir(i, fig1)
    i = i+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.

```
solv = Reoptimizer.TVizirGenetic()
# Name of the file that will contain
filename = "genetic.dump"
# whether = Test if a 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, filename):
    print("Restarting Vizir")
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

```
has_converged = opt.isConverged()
```

this argument being return as the results of the function.

The rest of this macro plays the role of the user in front of a ROOT-console (its python interplay of course). 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 !).

13.9.5.3 Graph

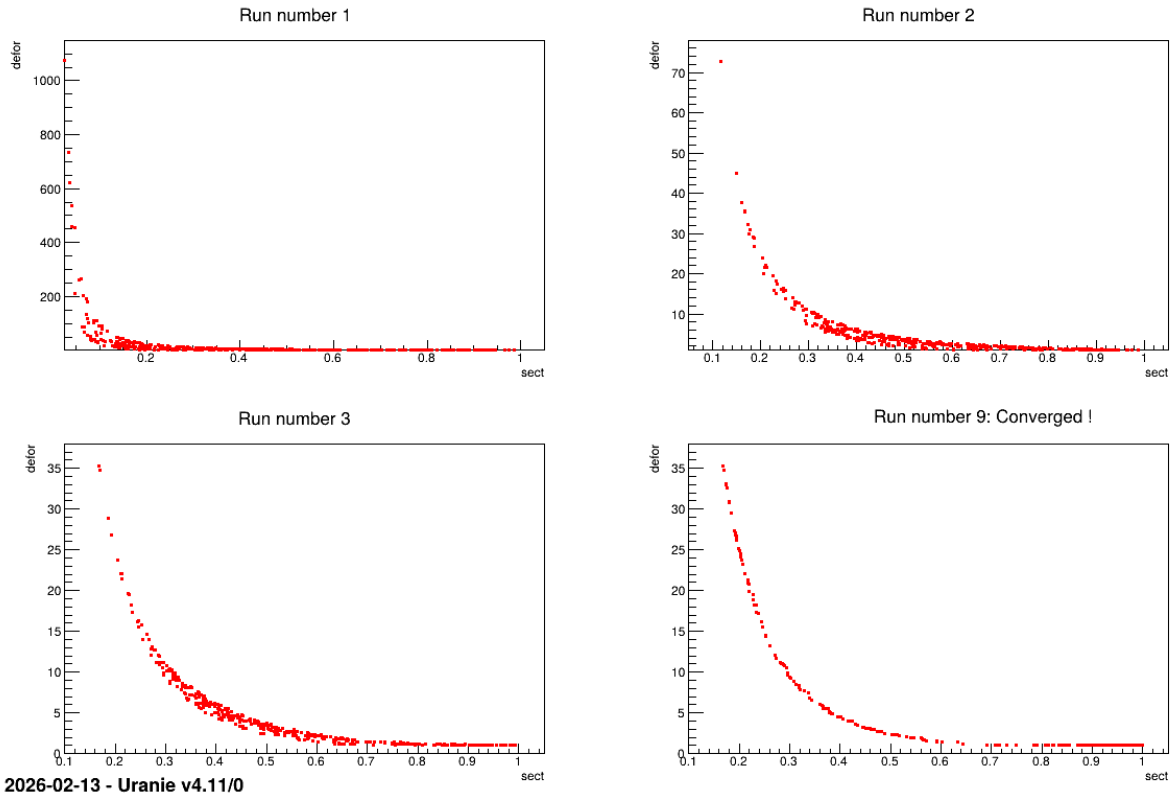


Figure 13.57: Graph of the macro **"reoptimizeHollowBarVizirSplitRuns.py"**

13.9.6 Macro "reoptimizeZoneBiSubMpi.py"

13.9.6.1 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.

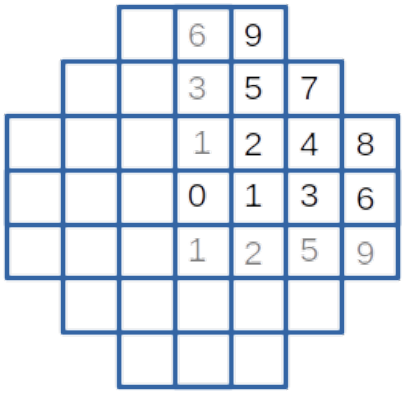


Figure 13.58: The core and its assemblies

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.

13.9.6.2 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

```
#!/usr/bin/env python


# 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
near1 = [
    [1, 1, 1, 1],  # 0
    [0, 2, 2, 3],
    [1, 1, 4, 5],
    [1, 4, 5, 6],  # 3
```

(continues on next page)

(continued from previous page)

```
    [2, 3, 7, 8],
    [2, 3, 7, 9],
    [3, 8, 9, 10],  # 6
    [4, 5, 10, 10],
    [4, 6, 10, 10],
    [5, 6, 10, 10]  # 9
]
# 4 secondary neighbours
near2 = [
    [2, 2, 2, 2],  # 0
    [1, 1, 4, 5],
    [0, 3, 3, 7],
    [2, 2, 8, 9],  # 3
    [1, 5, 6, 10],
    [1, 4, 6, 10],
    [4, 5, 10, 10],  # 6
    [2, 8, 9, 10],
    [3, 7, 10, 10],
    [3, 7, 10, 10]  # 9
]

# low level evaluation
def lowfun(edft, e_1, e_2, e_3, a_1, a_2, a_3):
    """ evaluate a zoning """
    def fill(loc, vid, val):
        """ fill a zone with an emission value """
        l_id = int(vid)
        loc[l_id] = val
        if l_id == 4:
            loc[5] = val

    def reception(loc, l_id):
        """ calculate reception in a zone """
        ret = loc[l_id]
        for l_iter in near1[l_id]:
            ret += loc[l_iter] / 8
        for l_iter in near2[l_id]:
            ret += loc[l_iter]/16
        return ret

    # init dft
    loc = [edft for i in range(11)]
    loc[8] = 0.8
    loc[9] = 0.8
    loc[10] = 0.0
    # init spec
    fill(loc, a_1, e_1)
    fill(loc, a_2, e_2)
    fill(loc, a_3, e_3)
    # normalize
    loc_all = loc[0] / 4
    for i in loc[1:]:
```

(continues on next page)

(continued from previous page)

```
        loc_all += i
    locn = [i*10/loc_all for i in loc]

    # max diff
    loc_max = reception(locn, 0)
    loc_min = loc_max
    for i_id in range(1, 10):
        next_var = reception(locn, i_id)
        if next_var < loc_min:
            loc_min = next_var
        elif next_var > loc_max:
            loc_max = next_var
    # output
    ret = [loc_max - loc_min]
    for i in locn[:-1]:
        ret.append(i)
    return ret
```

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.

```
#!/usr/bin/env python

import ROOT
import URANIE.DataServer as DataServer
import URANIE.Relauncher as Relauncher
import URANIE.Reoptimizer as Reoptimizer
import URANIE.MpiRelauncher as MpiRelauncher

from reoptimizeZoneCore import lowfun

# resume from tds
def tds_resume(tds, attr):
    """ get the best zoning from tds """
    def get_in_tds(tds, leaf, l_id):
        """ get a zoning objective """
        tds.GetTuple().GetEntry(l_id)
        return leaf.GetValue(0)

    siz = tds.GetTuple().GetEntries()
    leaves = [tds.GetTuple().GetLeaf(a.GetName()) for a in attr]

    # search min
    k = 0
    loc_min = get_in_tds(tds, leaves[0], 0)
    for i in range(1, siz):
        cur = get_in_tds(tds, leaves[0], i)
        if (cur < loc_min):
            loc_min = cur
```

(continues on next page)

(continued from previous page)

```
            k = i
    # all results
    tds.GetTuple().GetEntry(k)
    return [l.GetValue(0) for l in leaves]


# top level evaluation
def doefun(edft, ezon1, ezon2, ezon3) :
    """ find the best zoning for a given emission values """
    # inputs
    azon0 = DataServer.TAttribute("zon0", 0., 1.)
    azon1 = DataServer.TAttribute("zon1", 0., 1.)
    azon2 = DataServer.TAttribute("zon2", 0., 1.)
    azon3 = DataServer.TAttribute("zon3", 0., 1.)
    adef1 = DataServer.TAttribute("a1")
    adef2 = DataServer.TAttribute("a2")
    adef3 = DataServer.TAttribute("a3")
    funi = [azon0, azon1, azon2, azon3, adef1, adef2, adef3]
    # outputs
    namo = ["diff", "v0", "v1", "v2", "v3", "v4", "v5", "v6", "v7", "v8", "v9"]
    funo = [DataServer.TAttribute(n) for n in namo]
    # low fun
    fun = Relauncher.TPythonEval(lowfun)
    for inp in funi:
        fun.addInput(inp)
    for out in funo:
        fun.addOutput(out)
    # runner
    # run = Relauncher.TSequentialRun(fun)
    run = MpiRelauncher.TSubMpiRun(fun)
    run.startSlave()
    if run.onMaster():
        # doe def
        data = DataServer.TDataServer("doe", "tds4doe")
        data.keepFinalTuple(False)
        for i in funi[4:7]:
            data.addAttribute(i)
        data.fileDataRead("reoptimizeZoneDoe.dat", False, True, "quiet")
        # realize doe
        launch = Relauncher.TLauncher2(data, run)
        launch.addConstantValue(azon0, edft)
        launch.addConstantValue(azon1, ezon1)
        launch.addConstantValue(azon2, ezon2)
        launch.addConstantValue(azon3, ezon3)
        launch.solverLoop()
        # get objective
        ret = tds_resume(data, funo)
        # clean
        run.stopSlave()
        ROOT.SetOwnership(run, True)
        ROOT.SetOwnership(data, True)
        return ret
```

(continues on next page)

(continued from previous page)

```
    else:
        # return something to avoid warning
        # output number should be coherent
        return [0. for i in namo]


# top level control
def coeurR():
    """ optimization on emission values """
    # inputs
    nami = ["zone1", "zone2", "zone3", "zone4"]
    funi = [DataServer.TAttribute(n, 0., 1.) for n in nami]
    # outputs
    namo = ["diff", "v0", "v1", "v2", "v3", "v4", "v5", "v6", "v7", "v8", "v9"]
    funo = [DataServer.TAttribute(n) for n in namo]
    # top fun
    fun = Relauncher.TPythonEval(doefun)
    for inp in funi:
        fun.addInput(inp)
    for out in funo:
        fun.addOutput(out)
    fun.setMpi()
    # runner
    # run = Relauncher.TSequentialRun(fun)
    run = MpiRelauncher.TBiMpiRun(fun, 3)
    run.startSlave()
    if run.onMaster():
        # data
        data = DataServer.TDataServer("tdsvzr", "tds4optim")
        fun.addAllInputs(data)
        # optim
        gen = Reoptimizer.TVizirGenetic()
        gen.setSize(300, 200000, 100)
        viz = Reoptimizer.TVizir2(data, run, gen)
        viz.addObjective(funo[0])
        viz.solverLoop()
        # results
        data.exportData("__coeurPy__.dat")
        # clean
        run.stopSlave()
        ROOT.SetOwnership(run, True)


#ROOT.EnableThreadSafety()
coeurR()
```

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 `reoptimizeZoneBiSubMpi` 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 `reoptimizeZoneBiSubMpi` (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 python reoptimizeZoneBiSubMpi.py
```

At the begining of `reoptimizeZoneBiSubMpi` 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.

13.9.7 Macro "reoptimizeZoneBiFunMpi.py"

13.9.7.1 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.

> **Warning**
>
> To be run, the code need the mpi4py python module to be installed. This dependancy is not tested when Uranie is installed

13.9.7.2 Macro Uranie

The low level evaluation function is the same than in previous example and is not shown again.

```
#!/usr/bin/env python

import ROOT
import URANIE.DataServer as DataServer
import URANIE.Relauncher as Relauncher
import URANIE.Reoptimizer as Reoptimizer
import URANIE.MpiRelauncher as MpiRelauncher

from mpi4py import MPI
import ctypes

from reoptimizeZoneCore import lowfun

# mpi4py helpers
def getCalculMpiComm() :
```

(continues on next page)

(continued from previous page)

```
    """ convert C MPI_Comm to python """
    typemap = {
        ctypes.sizeof(ctypes.c_int) : ctypes.c_int, #mpich
        ctypes.sizeof(ctypes.c_void_p) : ctypes.c_void_p, #openmpi
    }
    comm = MPI.Comm()
    handle_t = typemap[MPI._sizeof(comm)]
    handle_new = handle_t.from_address(MPI._addressof(comm))
    handle_new.value = MpiRelauncher.TBiMpiRun.getCalculMpiCommPy()
    return comm


# doe
doe = [
    [0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 6], [4, 6, 7], [0, 6, 7], [0, 1, 7], # 7
    [0, 1, 3], [1, 2, 4], [2, 3, 6], [3, 4, 7], [0, 4, 6], [1, 6, 7], [0, 2, 7], # 14
    [0, 1, 4], [1, 2, 6], [2, 3, 7], [0, 3, 4], [1, 4, 6], [2, 6, 7], [0, 3, 7], # 21
    [0, 1, 6], [1, 2, 7], [0, 2, 3], [1, 3, 4], [2, 4, 6], [3, 6, 7], [0, 4, 7], # 28
    [0, 2, 4], [1, 3, 6], [2, 4, 7], [0, 3, 6], [1, 4, 7], [0, 2, 6], [1, 3, 7], # 35
]

# doe fun
def doefun(emz0, emz1, emz2, emz3) :
    """ find the best zoning for a given emission values """
    # mpi context
    comm = getCalculMpiComm()
    rank = comm.Get_rank()
    size = comm.Get_size()
    # search local best
    ind = rank
    idz1 = doe[ind][0]
    idz2 = doe[ind][1]
    idz3 = doe[ind][2]
    memo = lowfun(emz0, emz1, emz2, emz3, idz1, idz2, idz3)
    for next in range(rank, len(doe), size) :
        idz1 = doe[next][0]
        idz2 = doe[next][1]
        idz3 = doe[next][2]
        curr = lowfun(emz0, emz1, emz2, emz3, idz1, idz2, idz3)
        if curr[0] < memo[0] :
            memo = curr
    # global best
    # where is min
    ret = (memo[0], rank)
    (val, fid) = comm.allreduce(ret, op=MPI.MINLOC)
    # get all min value
    rval = memo
    if fid != 0 :
        if rank == fid :
            comm.send(memo, dest=0, tag=0)
        elif rank == 0 :
            rval = comm.recv(source=fid, tag=0)
```

(continues on next page)

(continued from previous page)

```
    return rval

# top level control
def coeurMpi():
    """ optimization on emission values """
    # inputs
    nami = ["zone1", "zone2", "zone3", "zone4"]
    funi = [DataServer.TAttribute(n, 0., 1.) for n in nami]
    # outputs
    namo = ["diff", "v0", "v1", "v2", "v3", "v4", "v5", "v6", "v7", "v8", "v9"]
    funo = [DataServer.TAttribute(n) for n in namo]
    # top fun
    fun = Relauncher.TPythonEval(doefun)
    for inp in funi:
        fun.addInput(inp)
    for out in funo:
        fun.addOutput(out)
    fun.setMpi()
    # runner
    # run = Relauncher.TSequentialRun(fun)
    run = MpiRelauncher.TBiMpiRun(fun, 3, False)
    run.startSlave()
    if run.onMaster():
        # data
        data = DataServer.TDataServer("tdsvzr", "tds4optim")
        fun.addAllInputs(data)
        # optim
        gen = Reoptimizer.TVizirGenetic()
        gen.setSize(300, 200000, 100)
        viz = Reoptimizer.TVizir2(data, run, gen)
        viz.addObjective(funo[0])
        viz.solverLoop()
        # results
        data.exportData("__coeurMpy__.dat")
        # clean
        run.stopSlave()
        #ROOT.SetOwnership(run, True)


#ROOT.EnableThreadSafety()
coeurMpi()
```

The top level function (`coeurMpi`) does not change from the previous example. The only difference is in the creation of the `TBiMpiRun` instances, which is done with an extra parameter set to False. This tell the constructor to avoid calling `MPI_Init` function. With mpi4py, this call is already done when the module is imported

The evaluation MPI fonction (`doefun`) is totaly different. It uses the function `getCalculMpiComm` to get the MPI Communicator object (`MPI.Comm`) dedicated to the calcul ressources. With it, different call to its method can be done : `Get_rank` and `Get_size` to get the context ; `allreduce`, `send` and `recv` to communicate between calculs ressources.

Function `getCalculMpiComm` is an helper function use to convert the MPI communicator get from Uranie `MpiRelauncher.TBiMpiRun.getCalculMpiCommPy` class method to the mpi4py equivalent object

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 many `TDataServer`.

13.10 Macros MetaModelOptim

13.10.1 Macro "metamodoptEgoHimmel.py"

13.10.1.1 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, 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 5 parameters and 8 optimal solutions.

13.10.1.2 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.

```
import ROOT

from ROOT.URANIE import DataServer as DataServer
from ROOT.URANIE import Relauncher as Relauncher
from ROOT.URANIE import MetaModelOptim as Ego

# user choice
krigingStep = 50
cpus = 6
optimStop = 360


# user optim fonction
def userfun(x1, x2, x3, x4, x5):
    ret = 0.
    tabx = [x1, x2, x3, x4, x5]
    tmp = x1*x1 - x2 - 6
    ret += tmp * tmp
    for i in range(1, 5):
        if i % 2:
            tmp = tabx[i]*tabx[i] - tabx[i-1] - 6
        else:
            tmp = tabx[i] - tabx[i-1]
        ret += tmp*tmp
    return [ret, ]


# optim procedure
def ego_test():
    # variables
    inputname = ["x1", "x2", "x3", "x4", "x5"]
    inputs = [DataServer.TUniformDistribution(str, -8., 8.) for str in inputname]
    output = DataServer.TAttribute("y")
```

(continues on next page)

(continued from previous page)

```
    # optim function
    fun = Relauncher.TPythonEval(userfun)
    for i in inputs:
        fun.addInput(i)
    fun.addOutput(output)

    # runner
    run = Relauncher.TThreadedRun(fun, cpus+1)
    run.startSlave()
    if (run.onMaster()):
        tds = DataServer.TDataServer("tds", "ego tds")
        for i in inputs:
            tds.addAttribute(i)

        kmod = Ego.TEgoKBModeler()
        kmod.setModel("matern7/2", "const", 1.0e-8)
        kmod.setSolver("ML", "Bobyqa", 300, 800)

        hjsolver = Ego.TEgoHjDynSolver()
        hjsolver.setSize(64, 16)

        egosolv = Ego.TEGO(tds, run)
        egosolv.setSize(krigingStep, optimStop)
        egosolv.setModeler(kmod)
        egosolv.setSolver(hjsolver)
        egosolv.addObjective(output)

        egosolv.solverLoop()

        tds.exportData("egoPy.dat")
        run.stopSlave()

# run test
ego_test()
```

At beginning, some constant values are defined.

- `cpus` value defines the number of threads that are used for evaluation.
- `krigingStep` and `optimStop` values define respectively 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 `ego_test` the optimisation to realize. The latter follows usual structure: It defines the problem variables, a `TPythonEval` 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). The `TDataServer` is empty, and a sampler is run implicitely to be able to create a first *surrogate model*. It may be unuseful when the `TDataServer` is filled by the `fileDataRead` method.

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 al-

gorithm. 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.

13.11 Macros Calibration

This section introduces a few examples of calibration in order to illustrate the different techniques introduced in *The Calibration module*, along with some of the available options.

13.11.1 Macro "calibrationMinimisationFlowrate1D.py"

13.11.1.1 Objective

The goal here is to calibrate the parameter H_l within the `flowrate` model, while varying only two inputs (r_ω and L). The remaining variables are fixed to the following values: $r = 25050$, $T_u = 89335$, $T_l = 89.55$, $H_u = 1050$, $K_\omega = 10950$. The context of this example has already been presented in *Use-case for this chapter*, including the model (implemented here as a C++ function) and the initial lines defining the `TDataServer` objects. This macro demonstrates how to apply a simple minimisation approach using a **Relauncher**-based architecture.

13.11.1.2 Macro Uranie

```
"""
Example of calibration using minimisation approach on flowrate 1D
"""
from URANIE import DataServer, Relauncher, Reoptimizer, Calibration
import ROOT

# Load the function flowrateCalib1D
ROOT.gROOT.LoadMacro("UserFunctions.C")

# Input reference file
ExpData = "Ex2DoE_n100_sd1.75.dat"

# define the reference
tdsRef = DataServer.TDataServer("tdsRef", "doe_exp_Re_Pr")
tdsRef.fileDataRead(ExpData)

# define the parameters
tdsPar = DataServer.TDataServer("tdsPar", "tdsPar")
tdsPar.addAttribute(DataServer.TAttribute("hl", 700.0, 760.0))
tdsPar.getAttribute("hl").setDefaultValue(728.0)

# Create the output attribute
out = DataServer.TAttribute("out")

# Create interface to assessors
Model = Relauncher.TCIntEval("flowrateCalib1D")
Model.addInput(tdsPar.getAttribute("hl"))
Model.addInput(tdsRef.getAttribute("rw"))
Model.addInput(tdsRef.getAttribute("l"))
```

(continues on next page)

(continued from previous page)

```
Model.addOutput(out)

# Set the runner
runner = Relauncher.TSequentialRun(Model)

# Set the calibration object
cal = Calibration.TMinimisation(tdsPar, runner, 1)
cal.setDistance("LS", tdsRef, "rw:l", "Qexp")
solv = Reoptimizer.TNloptSubplexe()
cal.setOptimProperties(solv)
cal.estimateParameters()

# Draw the residuals
canRes = ROOT.TCanvas("CanRes", "CanRes", 1200, 800)
padRes = ROOT.TPad("padRes", "padRes", 0, 0.03, 1, 1)
padRes.Draw()
padRes.cd()
cal.drawResiduals("Residuals", "*", "", "nonewcanvas")
```

Apart from the initial lines described in the section *Use-case for this chapter*, the key step is to define the starting point of the minimisation. This can be achieved either by calling the `setStartingPoint` method of the `TNlopt` class, or by assigning a default value to the parameter attributes. In this example, it is done as follows:

```
tdsPar.getAttribute("hl").setDefaultValue(728.0)
```

The model is defined (from a `TCIntEval` instance with the three input variables discussed above, in the **correct order**) along with the computation distribution method (sequential).

```
# Create interface to assessors
Model = Relauncher.TCIntEval("flowrateCalib1D")
Model.addInput(tdsPar.getAttribute("hl"))
Model.addInput(tdsRef.getAttribute("rw"))
Model.addInput(tdsRef.getAttribute("l"))
Model.addOutput(out)

# Set the runner
runner = Relauncher.TSequentialRun(Model)
```

Once this setup is complete, the calibration object (`TMinimisation`) is created. As explained in *Recommended distance and likelihood functions, construction method*, the first step is to define the distance function (here the least squares distance) using `setDistance`. This method also specifies the `TDataServer` containing the reference data, the names of the reference inputs, and the reference variable against which the model output is compared. Finally, the optimisation algorithm is defined by creating an instance of `TNloptSubplexe`, and the parameters are then estimated.

```
# Set the calibration object
cal = Calibration.TMinimisation(tdsPar, runner, 1)
cal.setDistance("LS", tdsRef, "rw:l", "Qexp")
solv = Reoptimizer.TNloptSubplexe()
cal.setOptimProperties(solv)
cal.estimateParameters()
```

The final part demonstrates how to display the results. Since this method produces a point estimate, only a single value is obtained, which is always shown on the screen, as illustrated in *Console*. Another important aspect is to examine

the residuals, as discussed in [Bla17]. This is illustrated in Figure 13.59, which shows the residuals of the *a posteriori* estimates, typically following a normal distribution.

```
# Draw the residuals
canRes = ROOT.TCanvas("CanRes", "CanRes", 1200, 800)
padRes = ROOT.TPad("padRes", "padRes", 0, 0.03, 1, 1)
padRes.Draw()
padRes.cd()
cal.drawResiduals("Residuals", "*", "", "nonewcanvas")
```

13.11.1.3 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

|....:....|....:....|....:....|....:
.*********************************************
*    Row   * tdsPar__n *     hl.hl * agreement *
*********************************************
*        0 *         0 * 749.72363 * 18.149368 *
*********************************************
```

13.11.1.4 Graph

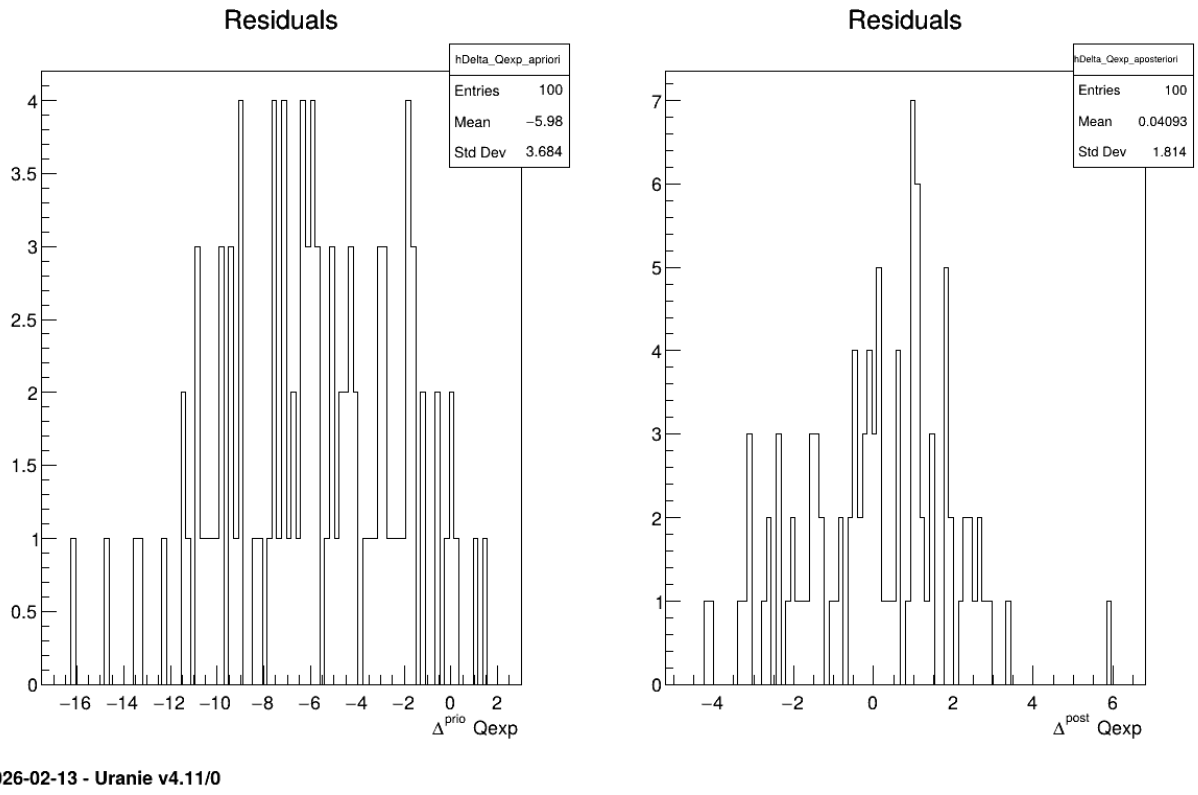


Figure 13.59: Residuals graph of the macro **"calibrationMinimisationFlowrate1D.py"**

13.11.2 Macro "calibrationMinimisationFlowrate2DVizir.py"

13.11.2.1 Objective

The goal here is to calibrate the parameters H_u and H_l within the `flowrateCalib2D` model (a two-dimensional version of the `flowrate` model), while varying only two inputs (r_ω and L). The remaining variables are fixed to the following values: $r = 25050$, $T_u = 89335$, $T_l = 89.55$, $K_\omega = 10950$. The context of this example has already been presented in *Use-case for this chapter*, including the model (implemented here as a C++ function) and the initial lines defining the `TDataServer` objects.

In addition to what has been presented in *Macro "calibrationMinimisationFlowrate1D.py"*, this macro introduces two new elements:

- the use of **Vizir** instead of a simpler `TNloptSolver`-inheriting instance;
- the discussion of problem identifiability, as introduced in [Bla17].

13.11.2.2 Macro Uranie

```
"""
Example of calibration through minimisation with Vizir in Flowrate 2D
"""
from URANIE import DataServer, Relauncher, Reoptimizer, Calibration
```

(continues on next page)

(continued from previous page)

```
import ROOT
# Load the function flowrateCalib2DVizir
ROOT.gROOT.LoadMacro("UserFunctions.C")

# Input reference file
ExpData = "Ex2DoE_n100_sd1.75.dat"

# define the reference
tdsRef = DataServer.TDataServer("tdsRef", "doe_exp_Re_Pr")
tdsRef.fileDataRead(ExpData)

# define the parameters
tdsPar = DataServer.TDataServer("tdsPar", "tdsPar")
tdsPar.addAttribute(DataServer.TAttribute("hu", 1020.0, 1080.0))
tdsPar.addAttribute(DataServer.TAttribute("hl", 720.0, 780.0))

# Create the output attribute
out = DataServer.TAttribute("out")

# Create interface to assessors
Model = Relauncher.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
runner = Relauncher.TSequentialRun(Model)

# Set the calibration object
cal = Calibration.TMinimisation(tdsPar, runner, 1)
cal.setDistance("relativeLS", tdsRef, "rw:l", "Qexp")
# Set optimisaiton properties
solv = Reoptimizer.TVizirGenetic()
solv.setSize(24, 15000, 100)
cal.setOptimProperties(solv)
# cal.getOptimMaster().setTolerance(1e-6)
cal.estimateParameters()

# Draw the residuals
canRes = ROOT.TCanvas("CanRes", "CanRes", 1200, 800)
padRes = ROOT.TPad("padRes", "padRes", 0, 0.03, 1, 1)
padRes.Draw()
padRes.cd()
cal.drawResiduals("Residuals", "*", "", "nonewcanvas")

# Draw the box plot of parameters
canPar = ROOT.TCanvas("CanPar", "CanPar", 1200, 800)
tdsPar.getTuple().SetMarkerStyle(20)
tdsPar.getTuple().SetMarkerSize(0.8)
tdsPar.Draw("hu:hl")
```

(continues on next page)

(continued from previous page)

```
# Look at the correlation and statistic
tdsPar.computeStatistic("hu:hl")
corr = tdsPar.computeCorrelationMatrix("hu:hl")
corr.Print()

print("hl is %3.8g +- %3.8g " % (tdsPar.getAttribute("hl").getMean(),
                                 tdsPar.getAttribute("hl").getStd()))
print("hu is %3.8g +- %3.8g " % (tdsPar.getAttribute("hu").getMean(),
                                 tdsPar.getAttribute("hu").getStd()))
```

Much of this code has already been covered in the previous section *Macro "calibrationMinimisationFlowrate1D.py"* (up to the sequential run). The main difference here is that the input parameter is now defined as a `TAttribute` with boundaries specifying the space in which the algorithm will search.

```
tdsPar.addAttribute(DataServer.TAttribute("hu", 1020.0, 1080.0))
tdsPar.addAttribute(DataServer.TAttribute("hl", 720.0, 780.0))
```

The model is defined (from a `TCIntEval` instance with the three input variables discussed above, in the **correct order**) along with the computation distribution method (sequential).

```
# Create interface to assessors
Model = Relauncher.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
runner = Relauncher.TSequentialRun(Model)
```

Once this setup is complete, the calibration object (`TMinimisation`) is created. As explained in *Recommended distance and likelihood functions, construction method*, the first step is to define the distance function (here the relative least squares distance) using `setDistance`. This method also specifies the `TDataServer` containing the reference data, the names of the reference inputs, and the reference variable against which the model output is compared. Finally, the optimisation algorithm is defined by creating an instance of `TVizirGenetic`, and the parameters are then estimated.

```
# Set the calibration object
cal = Calibration.TMinimisation(tdsPar, runner, 1)
cal.setDistance("relativeLS", tdsRef, "rw:l", "Qexp")
# Set optimisaiton properties
solv = Reoptimizer.TVizirGenetic()
solv.setSize(24, 15000, 100)
cal.setOptimProperties(solv)
# cal.getOptimMaster().setTolerance(1e-6)
cal.estimateParameters()
```

The final part demonstrates how to display the results. Since this method produces a point estimate, only a single value is obtained, which is always shown on the screen, as illustrated in *Console*. Another important aspect is to examine the residuals, as discussed in [Bla17]. This is illustrated in Figure 13.60, which shows the residuals of the *a posteriori* estimates, typically following a normal distribution. Finally, the parameter graph (Figure 13.61) reveals a wide variety of possible solutions. This highlights a problem of identifiability, since infinitely many parameter combinations

can lead to the same results, as further confirmed by the correlation matrix shown in *Console*.

```
# Draw the residuals
canRes = ROOT.TCanvas("CanRes", "CanRes", 1200, 800)
padRes = ROOT.TPad("padRes", "padRes", 0, 0.03, 1, 1)
padRes.Draw()
padRes.cd()
cal.drawResiduals("Residuals", "*", "", "nonewcanvas")

# Draw the box plot of parameters
canPar = ROOT.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")
corr = tdsPar.computeCorrelationMatrix("hu:hl")
corr.Print()

print("hl is %3.8g +- %3.8g " % (tdsPar.getAttribute("hl").getMean(),
                                 tdsPar.getAttribute("hl").getStd()))
print("hu is %3.8g +- %3.8g " % (tdsPar.getAttribute("hu").getMean(),
                                 tdsPar.getAttribute("hu").getStd()))
```

13.11.2.3 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                      Copyright (C) 2013-2026 CEA/DES
                      Contact: support-uranie@cea.fr
                      Date: Thu Feb 12, 2026

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 *
```

(continues on next page)

(continued from previous page)

```
*        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 752.4454 +- 19.946252
hu is 1052.4192 +- 19.959796
```

13.11.2.4 Graphs

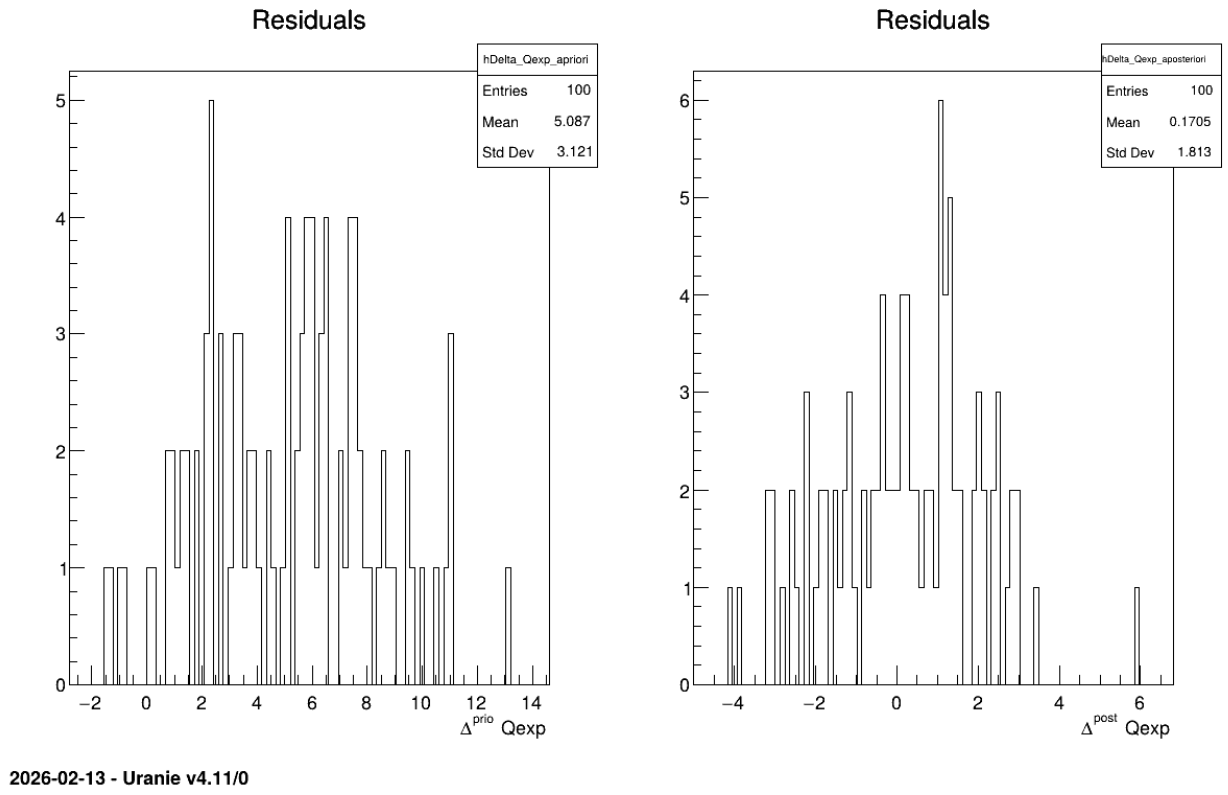


Figure 13.60: Residuals graph of the macro **"calibrationMinimisationFlowrate2DVizir.py"**

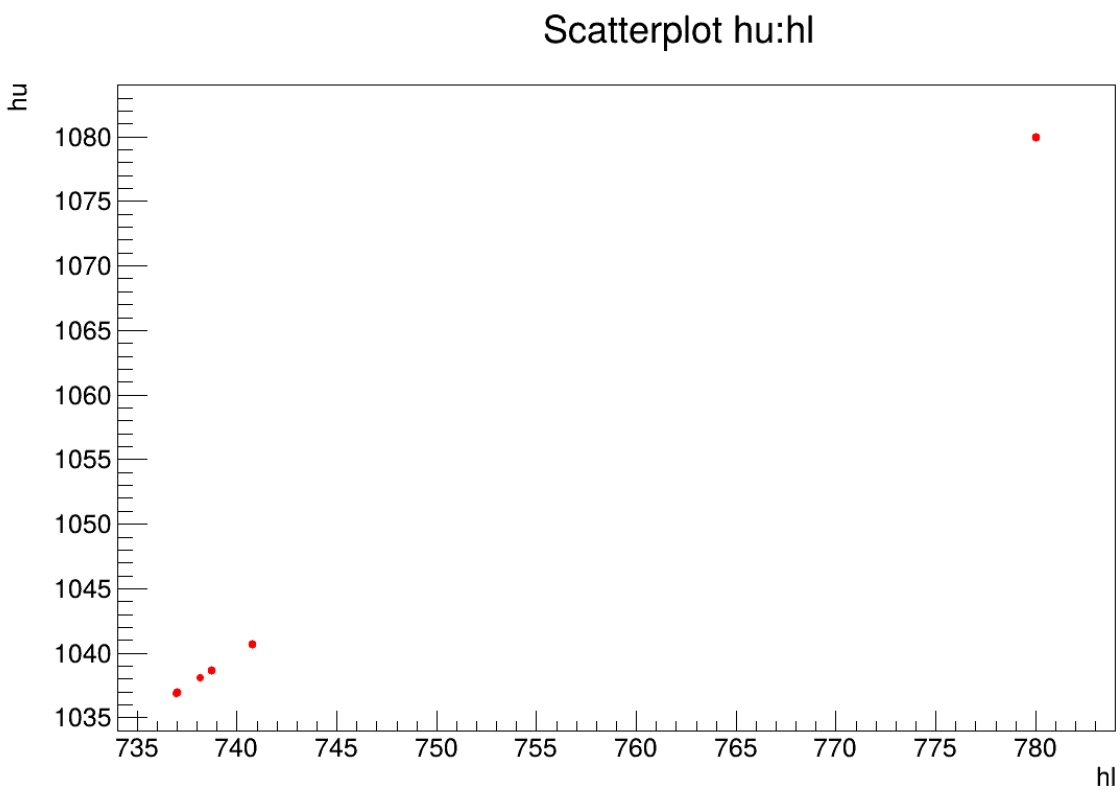


Figure 13.61: Parameter graph of the macro **"calibrationMinimisationFlowrate2DVizir.py"**

13.11.3 Macro "calibrationLinBayesFlowrate1D.py"

13.11.3.1 Objective

The goal here is to calibrate the parameter H_l within the `flowrate` model, while varying only two inputs (r_ω and L). The remaining variables are fixed to the following values: $r = 25050$, $T_u = 89335$, $T_l = 89.55$, $H_u = 1050$, $K_\omega = 10950$. The context of this example has already been presented in *Use-case for this chapter*, including the model (implemented here as a C++ function) and the initial lines defining the `TDataServer` objects. This macro demonstrates how to apply a linear Bayesian estimation technique using a **Relauncher**-based architecture.

13.11.3.2 Macro Uranie

```
"""
Example of linear bayesian calibration with simple 1D flowrate model
"""
from URANIE import DataServer, Relauncher, Calibration
import ROOT

# Load the function flowrateCalib1D
ROOT.gROOT.LoadMacro("UserFunctions.C")

# Input reference file
ExpData = "Ex2DoE_n100_sd1.75.dat"
```

(continues on next page)

(continued from previous page)

```
# define the reference
tdsRef = DataServer.TDataServer("tdsRef", "doe_exp_Re_Pr")
tdsRef.fileDataRead(ExpData)

# define the parameters
tdsPar = DataServer.TDataServer("tdsPar", "tdsPar")
tdsPar.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 760.0))

# Create the output attribute
out = DataServer.TAttribute("out")

# Create interface to assessors
Reg = Relauncher.TCIntEval("flowrateModelnoH")
Reg.addInput(tdsRef.getAttribute("rw"))
Reg.addInput(tdsRef.getAttribute("l"))
Reg.addOutput(DataServer.TAttribute("H"))
runnoH = Relauncher.TSequentialRun(Reg)
runnoH.startSlave()
if runnoH.onMaster():
    launch = Relauncher.TLauncher2(tdsRef, runnoH)
    launch.solverLoop()
    runnoH.stopSlave()

# Create interface to assessors
Model = Relauncher.TCIntEval("flowrateCalib1D")
Model.addInput(tdsPar.getAttribute("hl"))
Model.addInput(tdsRef.getAttribute("rw"))
Model.addInput(tdsRef.getAttribute("l"))
Model.addOutput(out)

# Set the runner
runner = Relauncher.TSequentialRun(Model)

# Set the covariance matrix of the input reference
sd = tdsRef.getValue("sd_eps", 0)
mat = ROOT.TMatrixD(100, 100)
for ival in range(tdsRef.getNPatterns()):
    mat[ival][ival] = (sd*sd)

# Set the calibration object
cal = Calibration.TLinearBayesian(tdsPar, runner, 1, "")
cal.setLikelihood("gauss_lin", tdsRef, "rw:l", "Qexp")
cal.setObservationCovarianceMatrix(mat)
cal.setRegressorName("H")
cal.setParameterTransformationFunction(ROOT.transf)
cal.estimateParameters()

# Draw the parameters
canPar = ROOT.TCanvas("CanPar", "CanPar", 1200, 800)
padPar = ROOT.TPad("padPar", "padPar", 0, 0.03, 1, 1)
padPar.Draw()
padPar.cd()
```

(continues on next page)

(continued from previous page)

```
cal.drawParameters("Parameters", "*", "nonewcanvas,transformed")

# Draw the residuals
canRes = ROOT.TCanvas("CanRes", "CanRes", 1200, 800)
padRes = ROOT.TPad("padRes", "padRes", 0, 0.03, 1, 1)
padRes.Draw()
padRes.cd()
cal.drawResiduals("Residuals", "*", "", "nonewcanvas")
```

Much of this code has already been covered in the previous section *Macro "calibrationMinimisationFlowrate1D.py"* (up to the sequential run). The main difference here is that the input parameter is now defined as a `TStochasticDistribution`, representing the *a priori* chosen distribution. In this example, it can be either a `TNormalDistribution` or a `TUniformDistribution` (see [Bla17]), with the latter being selected here:

```
tdsPar.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 760.0))
```

Another difference from the previous example is that the method must be slightly adapted to obtain values for the regressor. As discussed in [Bla17], linear Bayesian estimation can be applied when the model is approximately linear. This requires linearising the `flowrate` function, as demonstrated here:

$$f_\theta(x) = (2\pi T_u)\left(\ln(\frac{r}{r_\omega})\left[1+\frac{2LT_u}{\ln(\frac{r}{r_\omega})r_\omega^2 K_\omega}+\frac{T_u}{T_l}\right]\right)^{-1}\theta = H\times\theta$$

Here, 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 this expression, it is clear that we will be calibrating a newly defined parameter $\theta = (H_u - H_l)$, which must later be converted back into the parameter of interest. To obtain the regressor estimation, we simply use another C++ function, `flowrateModelnoH`, together with the standard **Relauncher** approach:

```
# Create interface to assessors
Reg = Relauncher.TCIntEval("flowrateModelnoH")
Reg.addInput(tdsRef.getAttribute("rw"))
Reg.addInput(tdsRef.getAttribute("l"))
Reg.addOutput(DataServer.TAttribute("H"))
runnoH = Relauncher.TSequentialRun(Reg)
runnoH.startSlave()
if runnoH.onMaster():
    launch = Relauncher.TLauncher2(tdsRef, runnoH)
    launch.solverLoop()
    runnoH.stopSlave()
```

This method also requires the input covariance matrix. The provided dataset (`Ex2DoE_n100_sd1.75.dat`) contains an estimate of the uncertainty affecting the reference measurements. Since this uncertainty is constant across all samples, the input covariance matrix is diagonal, with each diagonal element equal to the square of the standard deviation, as illustrated below:

```
# Set the covariance matrix of the input reference
sd = tdsRef.getValue("sd_eps", 0)
mat = ROOT.TMatrixD(100, 100)
for ival in range(tdsRef.getNPatterns()):
    mat[ival][ival] = (sd*sd)
```

The model is defined (from a `TCIntEval` instance with the three input variables discussed above, in the **correct order**) along with the computation distribution method (sequential).

The calibration object is then created using a Mahalanobis distance function, which is used here for illustrative purposes (see *Analytical linear Bayesian estimation* for more details). The three key steps are:

- Providing the input covariance matrix via the `setObservationCovarianceMatrix` method;
- Specifying the regressors' names using `setRegressorName`;
- Defining the parameter transformation function (optional) with `setParameterTransformationFunction`.

The final step is more involved: since we are calibrating θ, we need to recover the corresponding parameter value at the end. This is achieved by providing a C++ function that transforms the parameter estimated from the linearisation back into the parameter of interest. For illustration, this is implemented in `UserFunctions.C` via the `transf` function, shown below:

```
void transf(double *x, double *res)
{
    res[0] = 1050 - x[0];  // simply H_1 = \theta - H_u
}
```

The complete block of code discussed in this section is as follows:

```
# Set the calibration object
cal = Calibration.TLinearBayesian(tdsPar, runner, 1, "")
cal.setLikelihood("gauss_lin", tdsRef, "rw:1", "Qexp")
cal.setObservationCovarianceMatrix(mat)
cal.setRegressorName("H")
cal.setParameterTransformationFunction(ROOT.transf)
cal.estimateParameters()
```

The final part demonstrates how to display the results. Since this method produces normal *a posteriori* distributions, they are represented by a vector of means and a covariance structure, both easily accessible. The means are displayed on screen, as illustrated in *Console*. Two additional pieces of *a posteriori* information are presented as plots: the residuals (Figure 13.62), which show the expected normal distribution behavior, as discussed in [Bla17] and the posterior distribution (Figure 13.63).

```
# Draw the parameters
canPar = ROOT.TCanvas("CanPar", "CanPar", 1200, 800)
padPar = ROOT.TPad("padPar", "padPar", 0, 0.03, 1, 1)
padPar.Draw()
padPar.cd()
cal.drawParameters("Parameters", "*", "nonewcanvas,transformed")

# Draw the residuals
canRes = ROOT.TCanvas("CanRes", "CanRes", 1200, 800)
padRes = ROOT.TPad("padRes", "padRes", 0, 0.03, 1, 1)
padRes.Draw()
padRes.cd()
cal.drawResiduals("Residuals", "*", "", "nonewcanvas")
```

13.11.3.3 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

***  TLinearBayesian::computeParameters(); Transformed parameters estimated are

1x1 matrix is as follows

     |      0    |
------------------
   0 |      749.7
```

13.11.3.4 Graphs

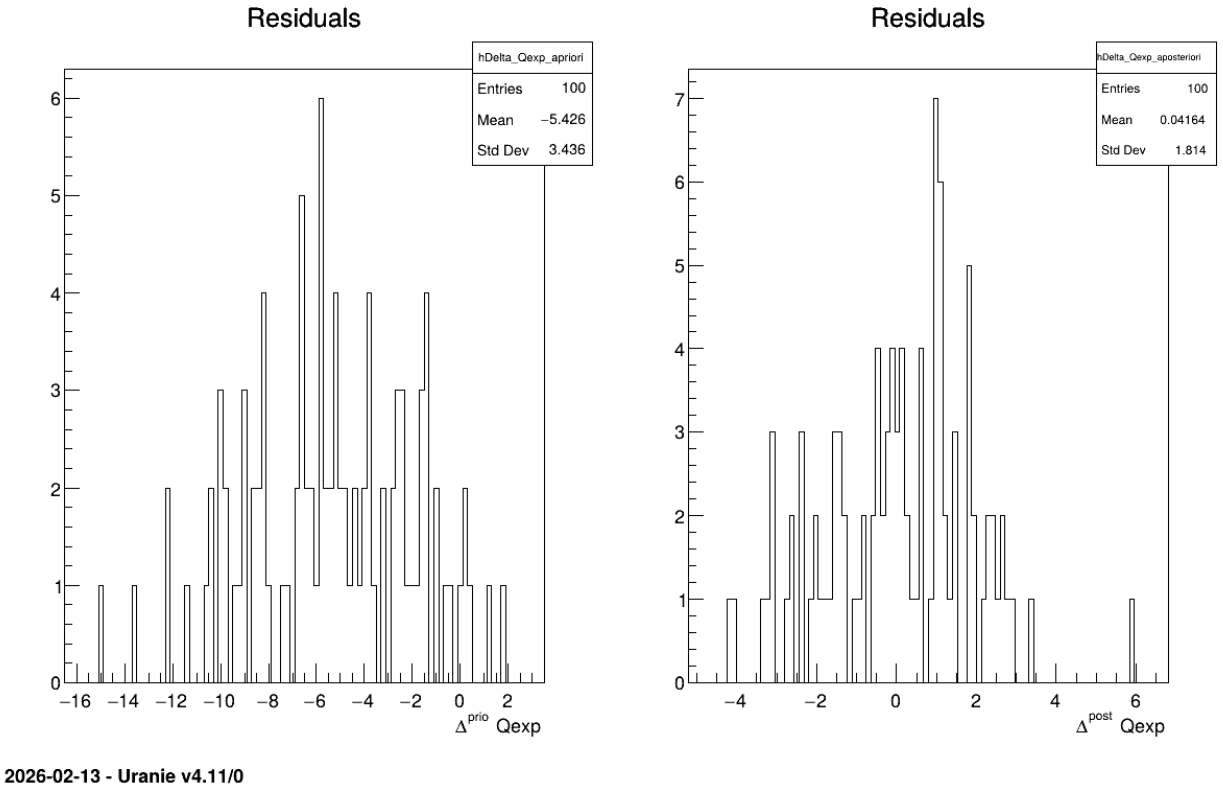


Figure 13.62: Residuals graph of the macro **"calibrationLinBayesFlowrate1D.py"**

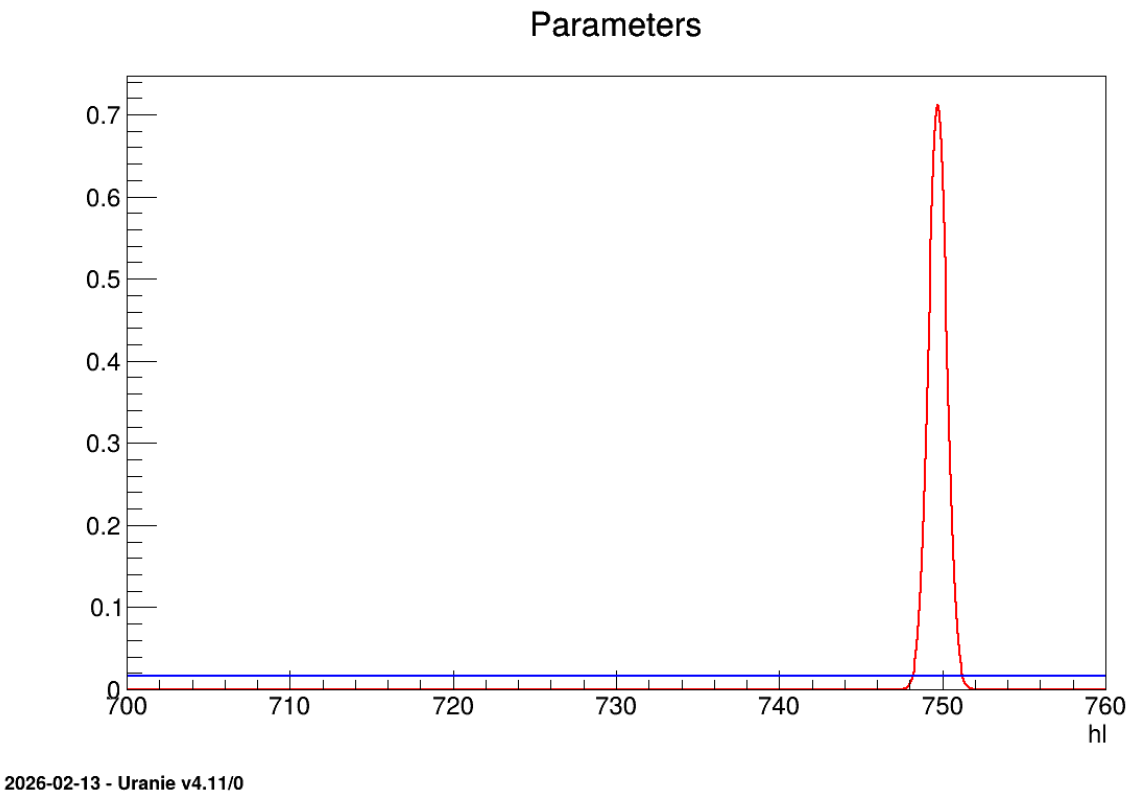


Figure 13.63: Parameter graph of the macro **"calibrationLinBayesFlowrate1D.py"**

13.11.4 Macro "calibrationRejectionABCFlowrate1D.py"

13.11.4.1 Objective

The goal here is to calibrate the parameter H_l within the `flowrate` model, while varying only two inputs (r_ω and L). The remaining variables are fixed to the following values: $r = 25050$, $T_u = 89335$, $T_l = 89.55$, $H_u = 1050$, $K_\omega = 10950$. The context of this example has already been presented in *Use-case for this chapter*, including the model (implemented here as a C++ function) and the initial lines defining the `TDataServer` objects. This macro demonstrates how to apply a rejection ABC method using a **Relauncher**-based architecture.

13.11.4.2 Macro Uranie

```
"""
Example of calibration using Rejection ABC approach on flowrate 1D
"""
from URANIE import DataServer, Calibration, Relauncher
import ROOT

# Load the function flowrateCalib1D
ROOT.gROOT.LoadMacro("UserFunctions.C")

# Input reference file
ExpData = "Ex2DoE_n100_sd1.75.dat"
```

(continues on next page)

(continued from previous page)

```
# define the reference
tdsRef = DataServer.TDataServer("tdsRef", "doe_exp_Re_Pr")
tdsRef.fileDataRead(ExpData)

# define the parameters
tdsPar = DataServer.TDataServer("tdsPar", "tdsPar")
tdsPar.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 760.0))

# Create the output attribute
out = DataServer.TAttribute("out")

# Create interface to assessors
Model = Relauncher.TCIntEval("flowrateCalib1D")
Model.addInput(tdsPar.getAttribute("hl"))
Model.addInput(tdsRef.getAttribute("rw"))
Model.addInput(tdsRef.getAttribute("l"))
Model.addOutput(out)

# Set the runner
runner = Relauncher.TSequentialRun(Model)

# Set the calibration object
nABC = 100
eps = 0.05
cal = Calibration.TRejectionABC(tdsPar, runner, nABC, "")
cal.setDistance("LS", tdsRef, "rw:l", "Qexp")
cal.setGaussianNoise("sd_eps")
cal.setPercentile(eps)
cal.estimateParameters()

# Draw the parameters
canPar = ROOT.TCanvas("CanPar", "CanPar", 1200, 800)
padPar = ROOT.TPad("padPar", "padPar", 0, 0.03, 1, 1)
padPar.Draw()
padPar.cd()
cal.drawParameters("Parameters", "*", "nonewcanvas")

# Draw the residuals
canRes = ROOT.TCanvas("CanRes", "CanRes", 1200, 800)
padRes = ROOT.TPad("padRes", "padRes", 0, 0.03, 1, 1)
padRes.Draw()
padRes.cd()
cal.drawResiduals("Residuals", "*", "", "nonewcanvas")

# Compute statistics
tdsPar.computeStatistic()
print("The mean of hl is %3.8g" % (tdsPar.getAttribute("hl").getMean()))
print("The std of hl is %3.8g" % (tdsPar.getAttribute("hl").getStd()))
```

Much of this code has already been covered in the previous section *Macro "calibrationMinimisationFlowrate1D.py"* (up to the sequential run). The main difference here is that the input parameter is now defined as a `TStochasticDistribution`, representing the *a priori* chosen distribution.

```
tdsPar.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 760.0))
```

The model is defined (from a `TCIntEval` instance with the three input variables discussed above, in the **correct order**) along with the computation distribution method (sequential).

The calibration object is then created by specifying both the number of elements in the final sample (`nABC = 100`) and the percentile of events retained (`eps = 0.05`, meaning that 2000 locations will be tested). Since the code is deterministic, uncertainty is introduced by adding random Gaussian noise. The standard deviation of this noise is defined on an event-by-event basis using a variable from the observation dataserver. Finally, the distance function is specified, and the estimation process is performed.

```
# Set the calibration object
nABC = 100
eps = 0.05
cal = Calibration.TRejectionABC(tdsPar, runner, nABC, "")
cal.setDistance("LS", tdsRef, "rw:1", "Qexp")
cal.setGaussianNoise("sd_eps")
cal.setPercentile(eps)
cal.estimateParameters()
```

The final part demonstrates how to display the results. Since this method produces samples of the *a posteriori* distributions, basic statistical information are directly displayed on screen, as shown in *Console*. Two additional pieces of *a posteriori* information are presented as plots: the residuals (Figure 13.64), which show the expected normal distribution behavior, as discussed in [Bla17] and the posterior distribution (Figure 13.65).

```
# Draw the parameters
canPar = ROOT.TCanvas("CanPar", "CanPar", 1200, 800)
padPar = ROOT.TPad("padPar", "padPar", 0, 0.03, 1, 1)
padPar.Draw()
padPar.cd()
cal.drawParameters("Parameters", "*", "nonewcanvas")

# Draw the residuals
canRes = ROOT.TCanvas("CanRes", "CanRes", 1200, 800)
padRes = ROOT.TPad("padRes", "padRes", 0, 0.03, 1, 1)
padRes.Draw()
padRes.cd()
cal.drawResiduals("Residuals", "*", "", "nonewcanvas")

# Compute statistics
tdsPar.computeStatistic()
print("The mean of hl is %3.8g" % (tdsPar.getAttribute("hl").getMean()))
print("The std of hl is %3.8g" % (tdsPar.getAttribute("hl").getStd()))
```

13.11.4.3 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
```

(continues on next page)

(continued from previous page)

```
 <URANIE::INFO> *** File[${SOURCEDIR}/meTIER/calibration/souRCE/
↪TDistanceLikelihoodFunction.cxx] Line[601]
 <URANIE::INFO> TDistanceLikelihoodFunction::setGaussianRandomNoise: gaussian random␣
↪noise(s) is added using information in [sd_eps] to modify the output variable(s)␣
↪[out].
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>
 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[${SOURCEDIR}/meTIER/calibration/souRCE/TRejectionABC.cxx]␣
↪Line[118]
 <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 749.92609
The std of hl is 1.9704229
```

13.11.4.4 Graphs

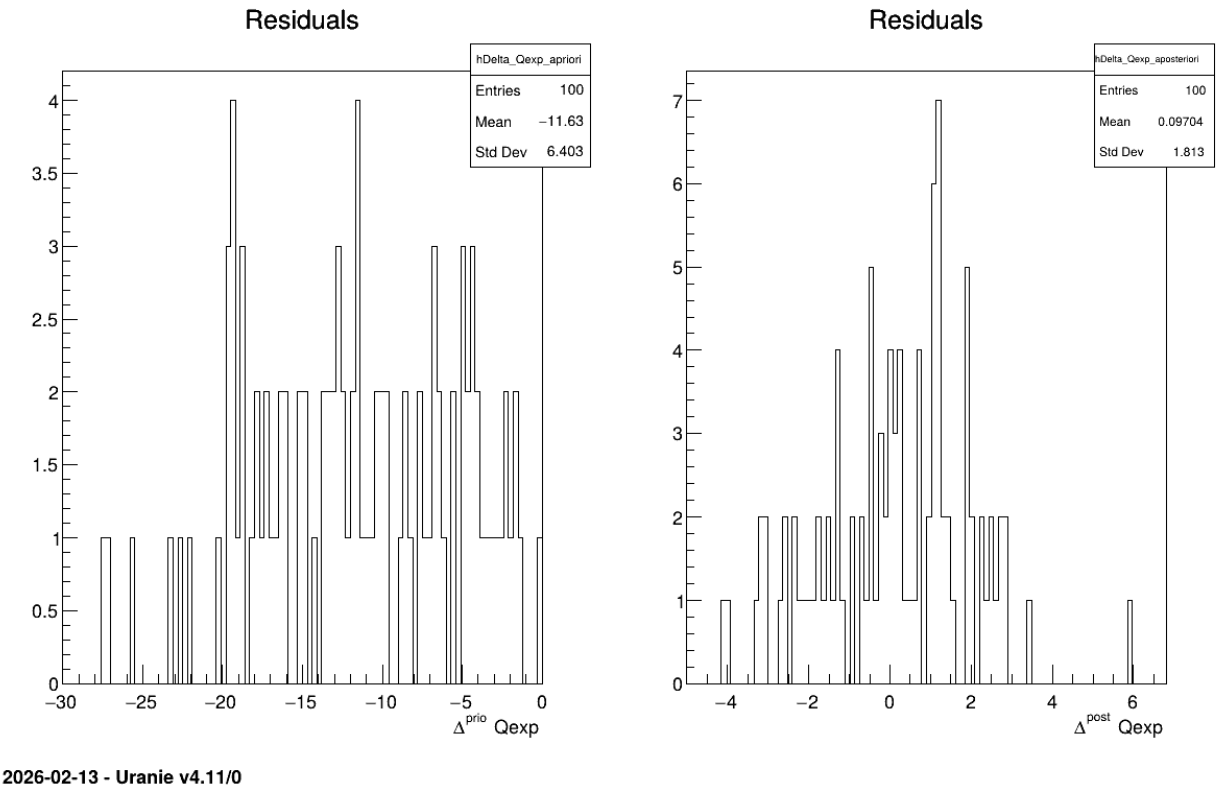


Figure 13.64: Residuals graph of the macro **"calibrationRejectionABCFlowrate1D.py"**

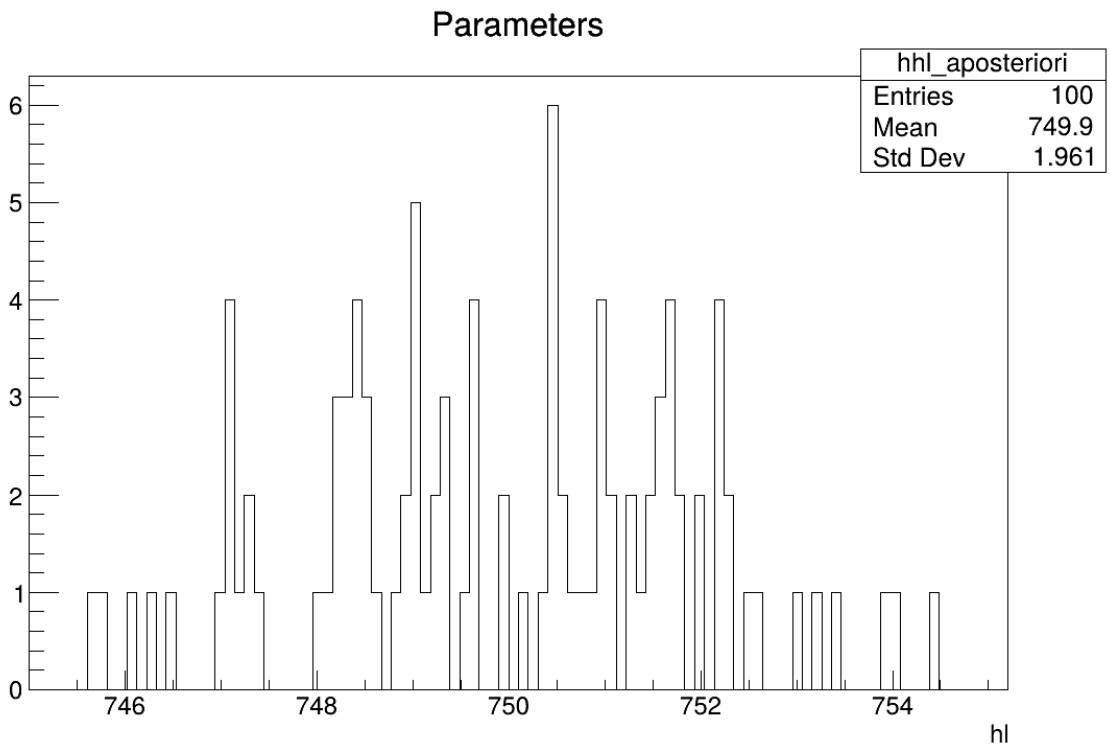


Figure 13.65: Parameter graph of the macro **"calibrationRejectionABCFlowrate1D.py"**

13.11.5 Macro "calibrationMCMCFlowrate1D.py"

13.11.5.1 Objective

The goal here is to calibrate the parameter H_l within the `flowrate` model, while varying only two inputs (r_ω and L). The remaining variables are fixed to the following values: $r = 25050$, $T_u = 89335$, $T_l = 89.55$, $H_u = 1050$, $K_\omega = 10950$. The context of this example has already been presented in *Use-case for this chapter*, including the model (implemented here as a C++ function) and the initial lines defining the `TDataServer` objects. This macro demonstrates how to apply the Markov chain Monte Carlo approach and more precisely the component-wise Metropolis-Hastings algorithm using a **Relauncher**-based architecture.

13.11.5.2 Macro Uranie

```
"""
Example of calibration using MCMC algo on flowrate 1D model
"""
from URANIE import DataServer, Relauncher, Calibration
import ROOT

# Load the function flowrateCalib1D
ROOT.gROOT.LoadMacro("UserFunctions.C")

# Input reference file$
ExpData = "Ex2DoE_n100_sd1.75.dat"
```

(continues on next page)

(continued from previous page)

```
# Define the reference
tdsRef = DataServer.TDataServer("tdsRef", "doe_exp_Re_Pr")
tdsRef.fileDataRead(ExpData)

# Define the parameters
tdsPar = DataServer.TDataServer("tdsPar", "tdsPar")
tdsPar.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 760.0))

# Create the output attribute
out = DataServer.TAttribute("out")

# Create interface to assessors
Model = Relauncher.TCIntEval("flowrateCalib1D")
Model.addInput(tdsPar.getAttribute("hl"))
Model.addInput(tdsRef.getAttribute("rw"))
Model.addInput(tdsRef.getAttribute("l"))
Model.addOutput(out)

# Set the runner
runner = Relauncher.TSequentialRun(Model)

# Set the calibration object
cal = Calibration.TMCMC(tdsPar, runner, 2000)

# Set the likelihood and reference with default gaussian log likelihood
cal.setLikelihood("log-gauss",tdsRef,"rw:l","Qexp","sd_eps")

# Set the MCMC algorithm to component-wise Metropolis-Hastings
cal.setAlgo("MH_1D")

# Set the acceptation ratio range between 20% and 50%
cal.setAcceptationRatioRange(0.2, 0.5)

# Set the number of iterations between two information displays
cal.setNbDump(500)

# Initialise 4 chains
cal.setMultistart(4)

# Set the starting points of each chain
cal.setStartingPoints(0, [740])
cal.setStartingPoints(1, [745])
cal.setStartingPoints(2, [750])
cal.setStartingPoints(3, [755])

# Set the standard deviation of the proposal for all the chains
cal.setProposalStd(-1, [5])

# Run the MCMC algorithm
cal.estimateParameters()
```

(continues on next page)

(continued from previous page)

```
# Quality assessment: Draw the trace of the Markov Chain
canTr = ROOT.TCanvas("CanTr","CanTr",1200,800)
padTr = ROOT.TPad("padTr","padTr",0, 0.03, 1, 1)
padTr.Draw()
padTr.cd()
cal.drawTrace("Trace", "*", "nonewcanvas")

# Quality assessment: Draw the acceptation ratio
canAcc = ROOT.TCanvas("CanAcc","CanAcc",1200,800)
padAcc = ROOT.TPad("padAcc","padAcc",0, 0.03, 1, 1)
padAcc.Draw()
padAcc.cd()
cal.drawAcceptationRatio("Acceptation ratio", "*", "nonewcanvas")

# Set the number of iterations to remove
burnin = 50
cal.setBurnin(burnin)

# Verify that the burn-in has been set correctely with Gelman-Rubin statistic
GelmanRubin_values = cal.diagGelmanRubin()

# Compute Effective Sample Size
ESS_values = cal.diagESS()

# Set the lag
lag = 1
cal.setLag(lag)

# Draw the residuals
canRes = ROOT.TCanvas("CanRes", "CanRes", 1200, 800)
padRes = ROOT.TPad("padRes", "padRes", 0, 0.03, 1, 1)
padRes.Draw()
padRes.cd()
cal.drawResiduals("Residuals", "*", "", "nonewcanvas")

# Draw the posterior distributions
canPar = ROOT.TCanvas("CanPar", "CanPar", 1200, 800)
padPar = ROOT.TPad("padPar", "padPar", 0, 0.03, 1, 1)
padPar.Draw()
padPar.cd()
cal.drawParameters("Posterior", "*", "nonewcanvas")
```

Much of this code has already been covered in the previous section *Macro "calibrationMinimisationFlowrate1D.py"* (up to the sequential run). The main difference here is that the input parameter is now defined as a `TStochasticDistribution`, representing the *a priori* chosen distribution.

```
tdsPar.addAttribute(DataServer.TUniformDistribution("hl", 700.0, 760.0))
```

The model is defined (from a `TCIntEval` instance with the three input variables discussed above, in the **correct order**) along with the computation distribution method (sequential).

The calibration object is then created by specifying both the number of iterations of the algorithm (set to 2000). The likelihood function is defined as Gaussian, with the reference dataserver, the input and output variables, and the standard

deviation of the experimental data provided as arguments. Several additional options are configured: the component-wise algorithm is selected, the acceptance range is set, the interval between two console displays is specified, the number of chains to initialise is defined, the starting points of each chain are assigned (one by one), and the standard deviation of the initial proposal distribution is set (for all chains at once). For further details on these options, see *Defining the TMCMC properties*. Finally, the estimation process is performed, using the chosen MCMC algorithm (component-wise Metropolis-Hastings).

```
# Set the calibration object
cal = Calibration.TMCMC(tdsPar, runner, 2000)

# Set the likelihood and reference with default gaussian log likelihood
cal.setLikelihood("log-gauss",tdsRef,"rw:1","Qexp","sd_eps")

# Set the MCMC algorithm to component-wise Metropolis-Hastings
cal.setAlgo("MH_1D")

# Set the acceptation ratio range between 20% and 50%
cal.setAcceptationRatioRange(0.2, 0.5)

# Set the number of iterations between two information displays
cal.setNbDump(500)

# Initialise 4 chains
cal.setMultistart(4)

# Set the starting points of each chain
cal.setStartingPoints(0, [740])
cal.setStartingPoints(1, [745])
cal.setStartingPoints(2, [750])
cal.setStartingPoints(3, [755])

# Set the standard deviation of the proposal for all the chains
cal.setProposalStd(-1, [5])

# Run the MCMC algorithm
cal.estimateParameters()
```

The final part demonstrates how to display and analyse the results. The first elements to examine are the trace plot (Figure 13.66) and the acceptance ratio plot (Figure 13.67). On these plots, you should check that the number of accepted iterations is sufficiently high (at least several hundred if the chains stabilise quickly — in our case, around 500), and that the converged acceptance ratio falls within a reasonable range (20–50% — here, it is close to 25%). When inspecting the trace, the chains should have converged around the same region (as they do here), and they should evolve rapidly, which indicates efficient exploration and low autocorrelation (also observed in our case). Finally, these plots help verify whether all chains behave stably and from which iteration they can be considered stable. This provides a first indication of a suitable burn-in value. In this example, the chains appear to have converged, and a burn-in value of 50 seems appropriate.

The burn-in size can then be defined and its consistency verified using the Gelman–Rubin statistic. In our case, a value of 1.00137 is reported in *Console* which is very close to 1 and confirms excellent convergence of the chains. Otherwise, a different burn-in value might have been more appropriate, the calculation could have been extended, or the MCMC algorithm settings adjusted.

The next step is to check for sample autocorrelation within the chains. The Effective Sample Size (ESS) provides a good indication of the lag to be used. In our case, the chains contain a reasonable number of independent samples (at least 257 each), and a lag of 1 is sufficient (see *Console*). Otherwise, the calculation could have been extended, or the MCMC algorithm settings adjusted (especially the standard deviation of the proposal distribution).

Two additional pieces of *a posteriori* information are presented as plots: the residuals (Figure 13.68), which show the expected normal distribution behavior, as discussed in [Bla17] and the posterior distribution (Figure 13.69).

```
# Quality assessment: Draw the trace of the Markov Chain
canTr = ROOT.TCanvas("CanTr","CanTr",1200,800)
padTr = ROOT.TPad("padTr","padTr",0, 0.03, 1, 1)
padTr.Draw()
padTr.cd()
cal.drawTrace("Trace", "*", "nonewcanvas")

# Quality assessment: Draw the acceptation ratio
canAcc = ROOT.TCanvas("CanAcc","CanAcc",1200,800)
padAcc = ROOT.TPad("padAcc","padAcc",0, 0.03, 1, 1)
padAcc.Draw()
padAcc.cd()
cal.drawAcceptationRatio("Acceptation ratio", "*", "nonewcanvas")

# Set the number of iterations to remove
burnin = 50
cal.setBurnin(burnin)

# Verify that the burn-in has been set correctely with Gelman-Rubin statistic
GelmanRubin_values = cal.diagGelmanRubin()

# Compute Effective Sample Size
ESS_values = cal.diagESS()

# Set the lag
lag = 1
cal.setLag(lag)

# Draw the residuals
canRes = ROOT.TCanvas("CanRes", "CanRes", 1200, 800)
padRes = ROOT.TPad("padRes", "padRes", 0, 0.03, 1, 1)
padRes.Draw()
padRes.cd()
cal.drawResiduals("Residuals", "*", "", "nonewcanvas")

# Draw the posterior distributions
canPar = ROOT.TCanvas("CanPar", "CanPar", 1200, 800)
padPar = ROOT.TPad("padPar", "padPar", 0, 0.03, 1, 1)
padPar.Draw()
padPar.cd()
cal.drawParameters("Posterior", "*", "nonewcanvas")
```

13.11.5.3 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

 <URANIE::INFO>
```

(continues on next page)

(continued from previous page)

```
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[/volatile/projet/uranie-ci/builds/Mfxi5pax-/0/uranie-cea/
↪uranie/souRCE/meTIER/calibration/souRCE/TMCMC.cxx] Line[193]
 <URANIE::INFO> TMCMC::constructor
 <URANIE::INFO> A folder named [MCMC_1] has been created to store the results of the␣
↪new calculation.
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>

 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[/volatile/projet/uranie-ci/builds/Mfxi5pax-/0/uranie-cea/
↪uranie/souRCE/meTIER/calibration/souRCE/TMCMC.cxx] Line[673]
 <URANIE::INFO> TMCMC::constructor
 <URANIE::INFO> MCMC_1_chain_0 file has been duplicated in the folder [MCMC_1] to␣
↪initiate [4] chains.
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>

500 events done
1000 events done
1500 events done
2000 events done
Computation finished with 2000 iterations.
Acceptation ratio [22.95%].
A posteriori mean coordinates are : (749.692)
500 events done
1000 events done
1500 events done
2000 events done
Computation finished with 2000 iterations.
Acceptation ratio [25.8%].
A posteriori mean coordinates are : (749.727)
500 events done
1000 events done
1500 events done
2000 events done
Computation finished with 2000 iterations.
Acceptation ratio [23.55%].
A posteriori mean coordinates are : (749.769)
500 events done
1000 events done
1500 events done
2000 events done
Computation finished with 2000 iterations.
Acceptation ratio [22.05%].
A posteriori mean coordinates are : (749.713)

Gelman-Rubin statistic for variable [hl] is equal to 1.00137. Excellent convergence.

Effective Sample Size (ESS) for variable [hl] and chain [0] is equal to 257.
Effective Sample Size (ESS) for variable [hl] and chain [1] is equal to 305.
```

(continues on next page)

(continued from previous page)

```
Effective Sample Size (ESS) for variable [hl] and chain [2] is equal to 405.
Effective Sample Size (ESS) for variable [hl] and chain [3] is equal to 371.

For uncorrelated samples, the lag should be set to 1.
```

13.11.5.4 Graphs

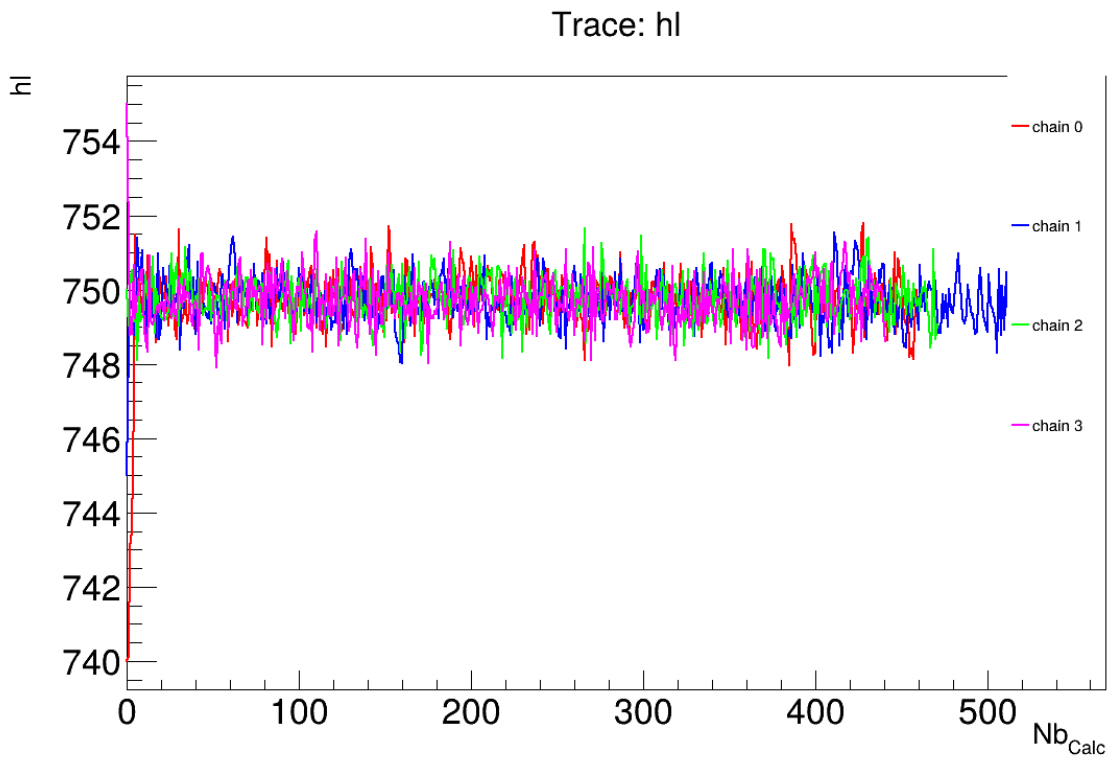


Figure 13.66: Trace graph of the macro **"calibrationMCMCFlowrate1D.py"**

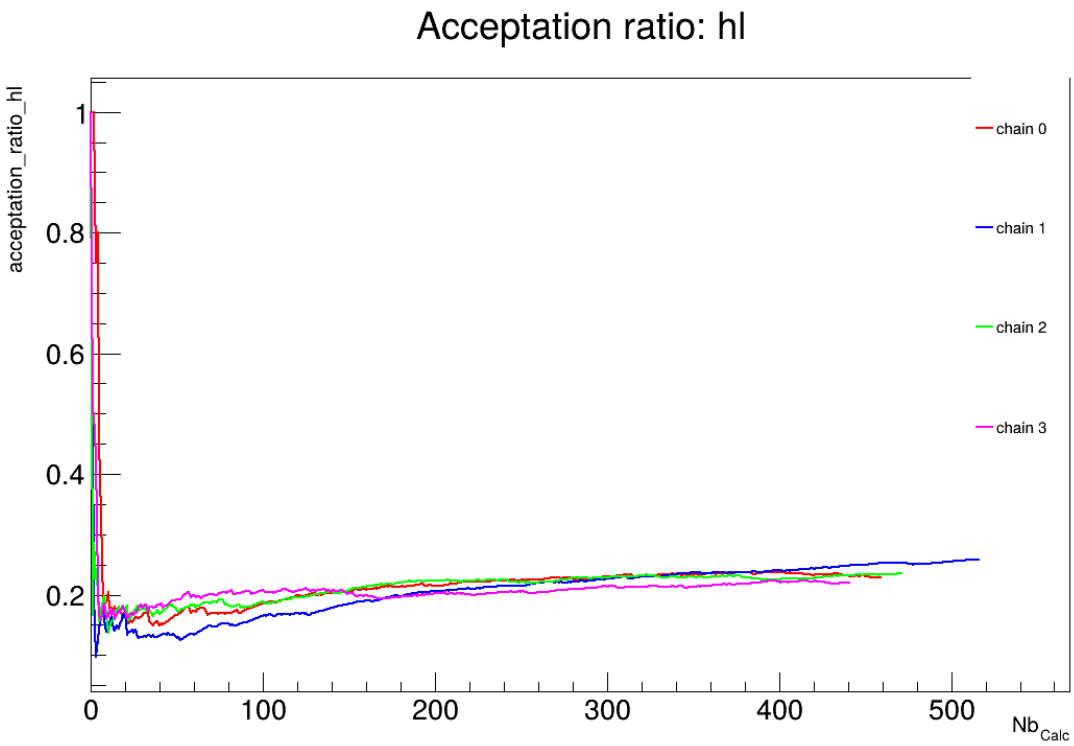


Figure 13.67: Acceptation ratio graph of the macro **"calibrationMCMCFlowrate1D.py"**

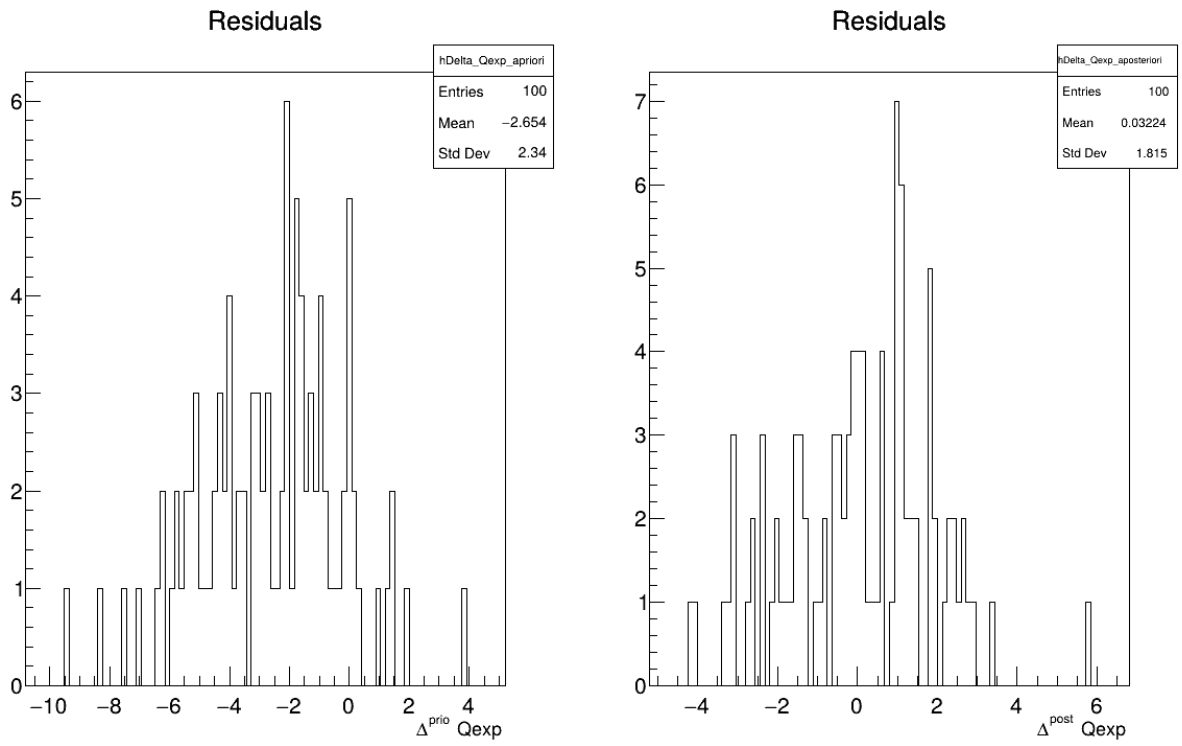


Figure 13.68: Residuals graph of the macro **"calibrationMCMCFlowrate1D.py"**

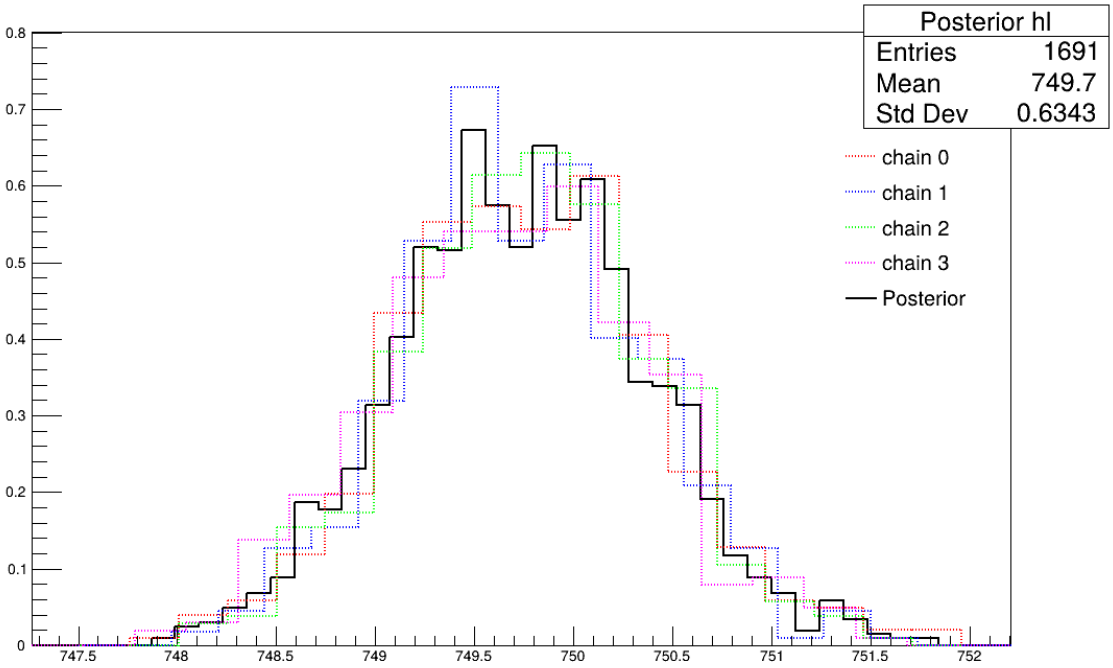


Figure 13.69: Parameter graph of the macro **"calibrationMCMCFlowrate1D.py"**

13.11.6 Macro "calibrationMCMCLinReg.py"

13.11.6.1 Objective

The goal here is to calibrate the parameters t_0 (constant term) and t_1 (multiplicative term) within a simple linear model (implemented in `UserFunctions.C`), against the observable outputs y_{Exp}, associated with given values of x stored in the input file `linReg_Database.dat`. The initial lines defining the `TDataServer` and the model are similar to those presented in *Use-case for this chapter*. This macro demonstrates how to apply the Markov chain Monte Carlo approach and more precisely the classic Metropolis-Hastings algorithm in two dimensions using a **Relauncher**-based architecture.

13.11.6.2 Macro Uranie

```
"""
Example of calibration using MCMC algo on linear model
"""
from URANIE import DataServer, Relauncher, Calibration
import ROOT

# Load the function Toy
ROOT.gROOT.LoadMacro("UserFunctions.C")

# Input reference file
ExpData = "linReg_Database.dat"
```

(continues on next page)

(continued from previous page)

```
# Define the reference
tdsRef = DataServer.TDataServer("tdsRef", "doe_exp_Re_Pr")
tdsRef.fileDataRead(ExpData)

# Define the uncertainty model guessing the standard deviation (here 0.2)
tdsRef.addAttribute("sd_eps", "0.2")

# Define the parameters
tdsPar = DataServer.TDataServer("tdsPar", "tdsPar")
binf_sea = -2.0
bsup_sea = 2.0
tdsPar.addAttribute(DataServer.TUniformDistribution("t0", binf_sea, bsup_sea))
tdsPar.addAttribute(DataServer.TUniformDistribution("t1", binf_sea, bsup_sea))

# Create the output attribute
out = DataServer.TAttribute("out")

# Create interface to assessors
myeval = Relauncher.TCIntEval("Toy")
myeval.addInput(tdsRef.getAttribute("x"))
myeval.addInput(tdsPar.getAttribute("t0"))
myeval.addInput(tdsPar.getAttribute("t1"))
myeval.addOutput(out)

# Set the runner
run = Relauncher.TSequentialRun(myeval)

# Set the calibration object with default Metropolis-Hastings algorithm
cal = Calibration.TMCMC(tdsPar, run, 4000)

# Set the likelihood and reference with default gaussian log likelihood
cal.setLikelihood("log-gauss", tdsRef, "x", "yExp", "sd_eps")

# Set the number of iterations between two information displays
cal.setNbDump(1000)

# Initialise 4 chains
cal.setMultistart(4)

# Set the starting points of each chain
cal.setStartingPoints(0, [0.5, -0.2])
cal.setStartingPoints(1, [-1.5, 1])
cal.setStartingPoints(2, [-1.2, -0.2])
cal.setStartingPoints(3, [0.3, 0.8])

# Set the standard deviation of the proposal for all the chains
cal.setProposalStd(-1, [0.05, 0.05])

# Run the MCMC algorithm
cal.estimateParameters()

# Quality assessment: Draw the trace of the Markov Chain
```

(continues on next page)

(continued from previous page)

```
canTr = ROOT.TCanvas("CanTr", "CanTr", 1200, 800)
padTr = ROOT.TPad("padTr", "padTr", 0, 0.03, 1, 1)
padTr.Draw()
padTr.cd()
cal.drawTrace("Trace", "*", "nonewcanvas")

# Quality assessment: Draw the acceptation ratio
canAcc = ROOT.TCanvas("CanAcc", "CanAcc", 1200, 800)
padAcc = ROOT.TPad("padAcc", "padAcc", 0, 0.03, 1, 1)
padAcc.Draw()
padAcc.cd()
cal.drawAcceptationRatio("Acceptation ratio", "*", "nonewcanvas")

# Set the number of iterations to remove
burnin = 100
cal.setBurnin(burnin)

# Verify that the burn-in has been set correctely with Gelman-Rubin statistic
GelmanRubin_values = cal.diagGelmanRubin()

# Compute Effective Sample Size
ESS_values = cal.diagESS()

# Set the lag
lag = 3
cal.setLag(lag)

# Quality assessment: Draw the 2D trace of the Markov Chain
canTr2D = ROOT.TCanvas("CanTr2D", "CanTr2D", 1200, 800)
padTr2D = ROOT.TPad("padTr2D", "padTr2D", 0, 0.03, 1, 1)
padTr2D.Draw()
padTr2D.cd()
cal.draw2DTrace("Trace 2D", "t0:t1", "nonewcanvas")

# Draw the residuals
canRes = ROOT.TCanvas("CanRes", "CanRes", 1200, 800)
padRes = ROOT.TPad("padRes", "padRes", 0, 0.03, 1, 1)
padRes.Draw()
padRes.cd()
cal.drawResiduals("Residuals", "*", "", "nonewcanvas")

# Draw the posterior distributions
canPar = ROOT.TCanvas("CanPar","CanPar",1200,800)
padPar = ROOT.TPad("padPar","padPar",0, 0.03, 1, 1)
padPar.Draw()
padPar.cd()
cal.drawParameters("Posterior", "*", "nonewcanvas")
```

Much of this code has already been covered in the previous section *Macro "calibrationMinimisationFlowrate1D.py"* (up to the sequential run). The main difference here is that the input parameter is now defined as a `TStochasticDistribution`, representing the *a priori* chosen distribution.

Another difference from the previous example *Macro "calibrationMCMCFlowrate1D.py"* is that no uncertainty associated

with the observables is provided in the input file. To this end, a new attribute is defined and added to the reference dataserver, with an assumed standard deviation of y_{Exp} equal to 0.2.

```
# Define the uncertainty model guessing the standard deviation (here 0.2)
tdsRef.addAttribute("sd_eps", "0.2")

# Define the parameters
tdsPar = DataServer.TDataServer("tdsPar", "tdsPar")
binf_sea = -2.0
bsup_sea = 2.0
tdsPar.addAttribute(DataServer.TUniformDistribution("t0", binf_sea, bsup_sea))
tdsPar.addAttribute(DataServer.TUniformDistribution("t1", binf_sea, bsup_sea))
```

The model is defined (from a `TCIntEval` instance with the three input variables discussed above, in the **correct order**) along with the computation distribution method (sequential).

The calibration object is then created by specifying both the number of iterations of the algorithm (set to 4000). The likelihood function is defined as Gaussian, with the reference dataserver, the input and output variables, and the assumed standard deviation of the experimental data (see previous paragraph) provided as arguments. Several additional options are configured: the interval between two console displays is specified, the number of chains to initialise is defined, the starting points of each chain are assigned (one by one), and the standard deviation of the initial proposal distribution is set (for all chains at once). For further details on these options, see *Defining the TMCMC properties*. Finally, the estimation process is performed, using the default MCMC algorithm (classic Metropolis-Hastings).

```
# Set the calibration object with default Metropolis-Hastings algorithm
cal = Calibration.TMCMC(tdsPar, run, 4000)

# Set the likelihood and reference with default gaussian log likelihood
cal.setLikelihood("log-gauss", tdsRef, "x", "yExp", "sd_eps")

# Set the number of iterations between two information displays
cal.setNbDump(1000)

# Initialise 4 chains
cal.setMultistart(4)

# Set the starting points of each chain
cal.setStartingPoints(0, [0.5, -0.2])
cal.setStartingPoints(1, [-1.5, 1])
cal.setStartingPoints(2, [-1.2, -0.2])
cal.setStartingPoints(3, [0.3, 0.8])

# Set the standard deviation of the proposal for all the chains
cal.setProposalStd(-1, [0.05, 0.05])

# Run the MCMC algorithm
cal.estimateParameters()
```

The final part demonstrates how to display and analyse the results. The first elements to examine are the trace plot (Figure 13.70) and the acceptance ratio plot (Figure 13.71). On these plots, you should check that the number of accepted iterations is sufficiently high (at least several hundred if the chains stabilise quickly — in our case, around 1500), and that the converged acceptance ratio falls within a reasonable range (20–50% — here, it is close to 40%). When inspecting the trace, the chains should have converged around the same region (as they do here), and they should evolve rapidly, which indicates efficient exploration and low autocorrelation (also observed in our case). Finally, these plots help verify whether all chains behave stably and from which iteration they can be considered stable. This provides a first indication of a suitable

burn-in value. In this example, the chains appear to have converged, and a burn-in value of 100 seems appropriate.

The burn-in size can then be defined and its consistency verified using the Gelman–Rubin statistic. In our case, values of 1.00092 and 0.99993 (for parameters t_0 and t_1 respectively) are reported in *Console* which are very close to 1 and confirms excellent convergence of the chains for both parameters. Otherwise, a different burn-in value might have been more appropriate, the calculation could have been extended, or the MCMC algorithm settings adjusted.

The next step is to check for sample autocorrelation within the chains. The Effective Sample Size (ESS) provides a good indication of the lag to be used. In our case, the chains contain a reasonable number of independent samples (at least 424 each), and a lag of 3 is sufficient (see *Console*). Otherwise, the calculation could have been extended, or the MCMC algorithm settings adjusted (especially the standard deviation of the proposal distribution).

It is also possible to represent the chains in two dimensions (Figure 13.72). This provides an alternative way to verify whether the chains converge to the same region and highlights potential covariance between the posterior distributions of the two parameters.

Two additional pieces of *a posteriori* information are presented as plots: the residuals (Figure 13.73), which show the expected normal distribution behavior, as discussed in [Bla17] and the posterior distribution (Figure 13.74).

```
# Quality assessment: Draw the trace of the Markov Chain
canTr = ROOT.TCanvas("CanTr", "CanTr", 1200, 800)
padTr = ROOT.TPad("padTr", "padTr", 0, 0.03, 1, 1)
padTr.Draw()
padTr.cd()
cal.drawTrace("Trace", "*", "nonewcanvas")

# Quality assessment: Draw the acceptation ratio
canAcc = ROOT.TCanvas("CanAcc", "CanAcc", 1200, 800)
padAcc = ROOT.TPad("padAcc", "padAcc", 0, 0.03, 1, 1)
padAcc.Draw()
padAcc.cd()
cal.drawAcceptationRatio("Acceptation ratio", "*", "nonewcanvas")

# Set the number of iterations to remove
burnin = 100
cal.setBurnin(burnin)

# Verify that the burn-in has been set correctely with Gelman-Rubin statistic
GelmanRubin_values = cal.diagGelmanRubin()

# Compute Effective Sample Size
ESS_values = cal.diagESS()

# Set the lag
lag = 3
cal.setLag(lag)

# Quality assessment: Draw the 2D trace of the Markov Chain
canTr2D = ROOT.TCanvas("CanTr2D", "CanTr2D", 1200, 800)
padTr2D = ROOT.TPad("padTr2D", "padTr2D", 0, 0.03, 1, 1)
padTr2D.Draw()
padTr2D.cd()
cal.draw2DTrace("Trace 2D", "t0:t1", "nonewcanvas")

# Draw the residuals
```

(continues on next page)

(continued from previous page)

```
canRes = ROOT.TCanvas("CanRes", "CanRes", 1200, 800)
padRes = ROOT.TPad("padRes", "padRes", 0, 0.03, 1, 1)
padRes.Draw()
padRes.cd()
cal.drawResiduals("Residuals", "*", "", "nonewcanvas")

# Draw the posterior distributions
canPar = ROOT.TCanvas("CanPar","CanPar",1200,800)
padPar = ROOT.TPad("padPar","padPar",0, 0.03, 1, 1)
padPar.Draw()
padPar.cd()
cal.drawParameters("Posterior", "*", "nonewcanvas")
```

13.11.6.3 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[/volatile/projet/uranie-ci/builds/Mfxi5pax-/0/uranie-cea/
↪uranie/souRCE/meTIER/calibration/souRCE/TMCMC.cxx] Line[193]
 <URANIE::INFO> TMCMC::constructor
 <URANIE::INFO> A folder named [MCMC_1] has been created to store the results of the␣
↪new calculation.
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>

 <URANIE::INFO>
 <URANIE::INFO> *** URANIE INFORMATION ***
 <URANIE::INFO> *** File[/volatile/projet/uranie-ci/builds/Mfxi5pax-/0/uranie-cea/
↪uranie/souRCE/meTIER/calibration/souRCE/TMCMC.cxx] Line[673]
 <URANIE::INFO> TMCMC::constructor
 <URANIE::INFO> MCMC_1_chain_0 file has been duplicated in the folder [MCMC_1] to␣
↪initiate [4] chains.
 <URANIE::INFO> *** END of URANIE INFORMATION ***
 <URANIE::INFO>

1000 events done
2000 events done
3000 events done
4000 events done
Computation finished with 4000 iterations.
Acceptation ratio [36.325%].
A posteriori mean coordinates are : (-0.433531,0.369056)
1000 events done
2000 events done
3000 events done
4000 events done
```

(continues on next page)

(continued from previous page)

```
Computation finished with 4000 iterations.
Acceptation ratio [36.625%].
A posteriori mean coordinates are : (-0.455715,0.378325)
1000 events done
2000 events done
3000 events done
4000 events done
Computation finished with 4000 iterations.
Acceptation ratio [35.475%].
A posteriori mean coordinates are : (-0.452614,0.368962)
1000 events done
2000 events done
3000 events done
4000 events done
Computation finished with 4000 iterations.
Acceptation ratio [37.925%].
A posteriori mean coordinates are : (-0.440476,0.376135)

Gelman-Rubin statistic for variable [t0] is equal to 1.00092. Excellent convergence.
Gelman-Rubin statistic for variable [t1] is equal to 0.99993. Excellent convergence.

Effective Sample Size (ESS) for variable [t0] and chain [0] is equal to 426.
Effective Sample Size (ESS) for variable [t1] and chain [0] is equal to 893.
Effective Sample Size (ESS) for variable [t0] and chain [1] is equal to 424.
Effective Sample Size (ESS) for variable [t1] and chain [1] is equal to 727.
Effective Sample Size (ESS) for variable [t0] and chain [2] is equal to 471.
Effective Sample Size (ESS) for variable [t1] and chain [2] is equal to 594.
Effective Sample Size (ESS) for variable [t0] and chain [3] is equal to 440.
Effective Sample Size (ESS) for variable [t1] and chain [3] is equal to 802.

For uncorrelated samples, the lag should be set to 3.
```

13.11.6.4 Graphs

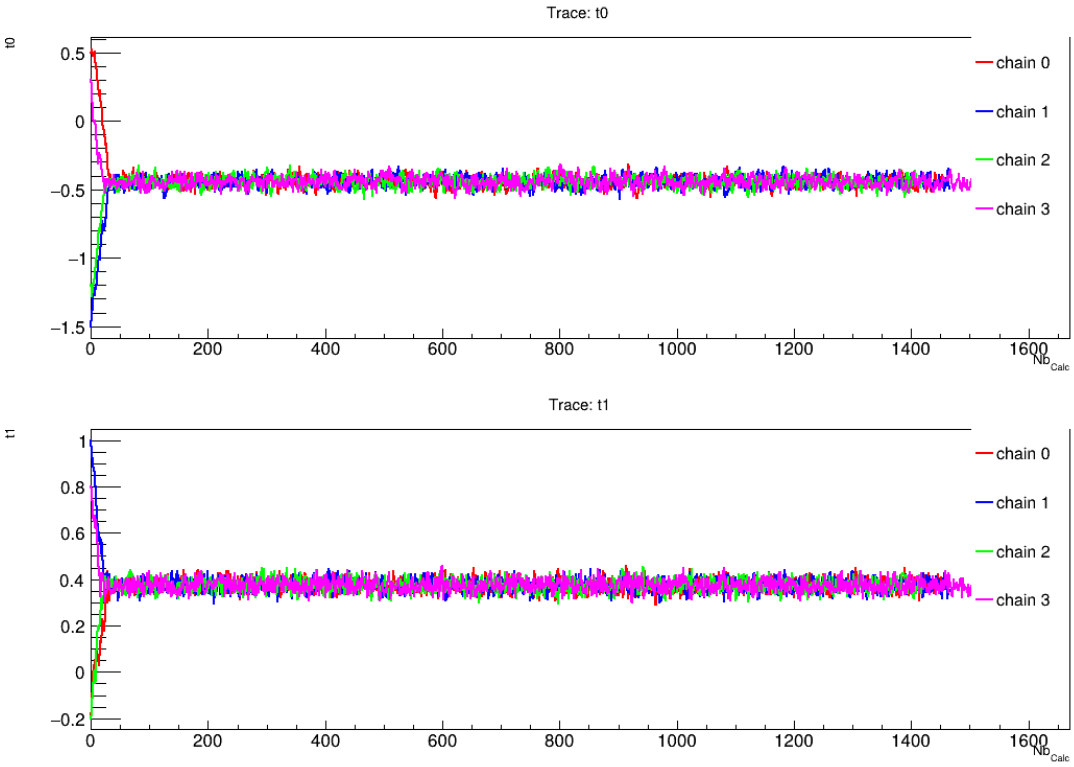


Figure 13.70: Trace graph of the macro **"calibrationMCMCLinReg.py"**

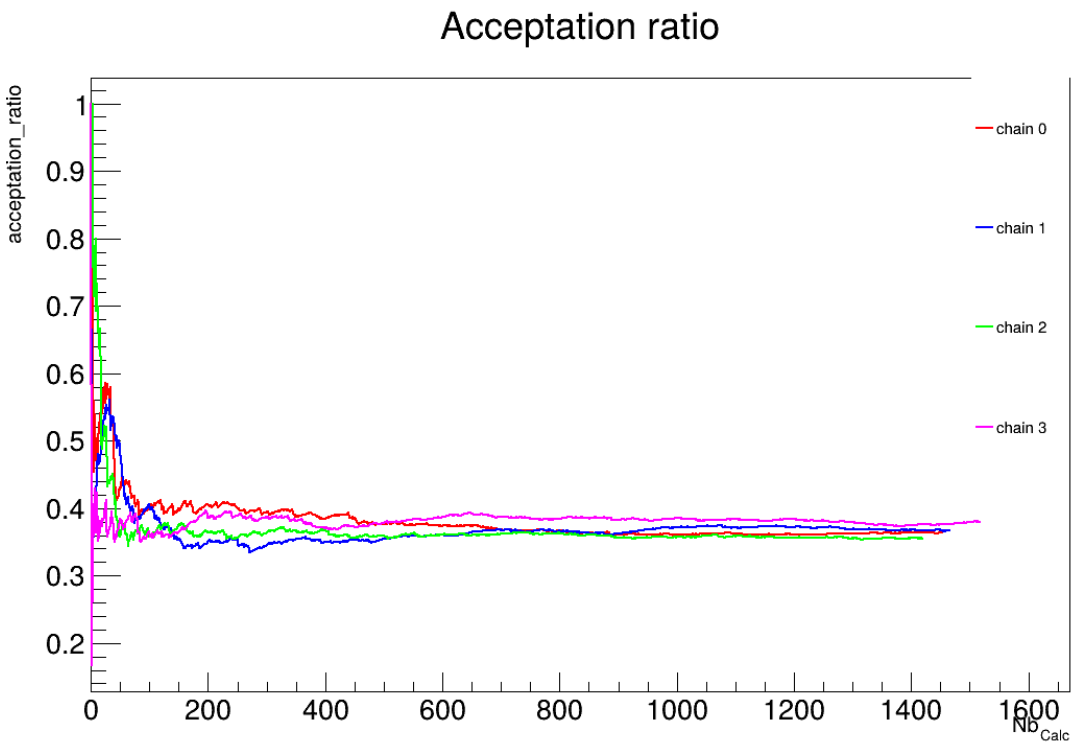


Figure 13.71: Acceptation ratio graph of the macro **"calibrationMCMCLinReg.py"**

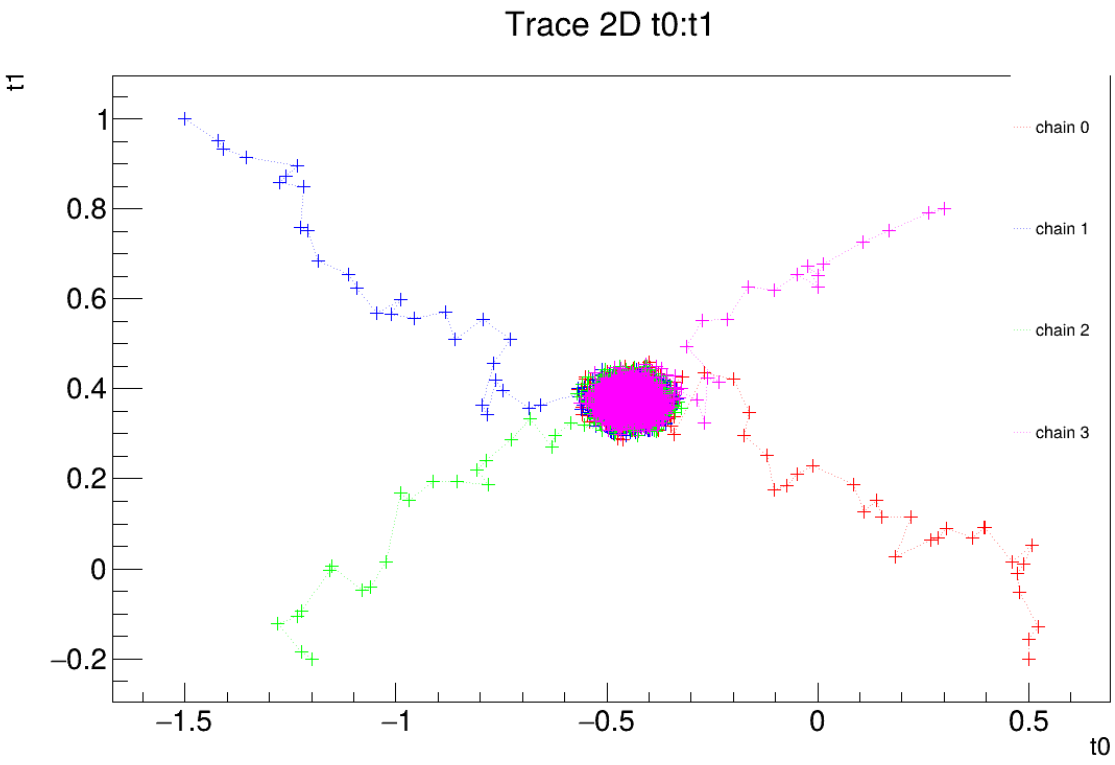


Figure 13.72: 2D Trace graph of the macro **"calibrationMCMCLinReg.py"**

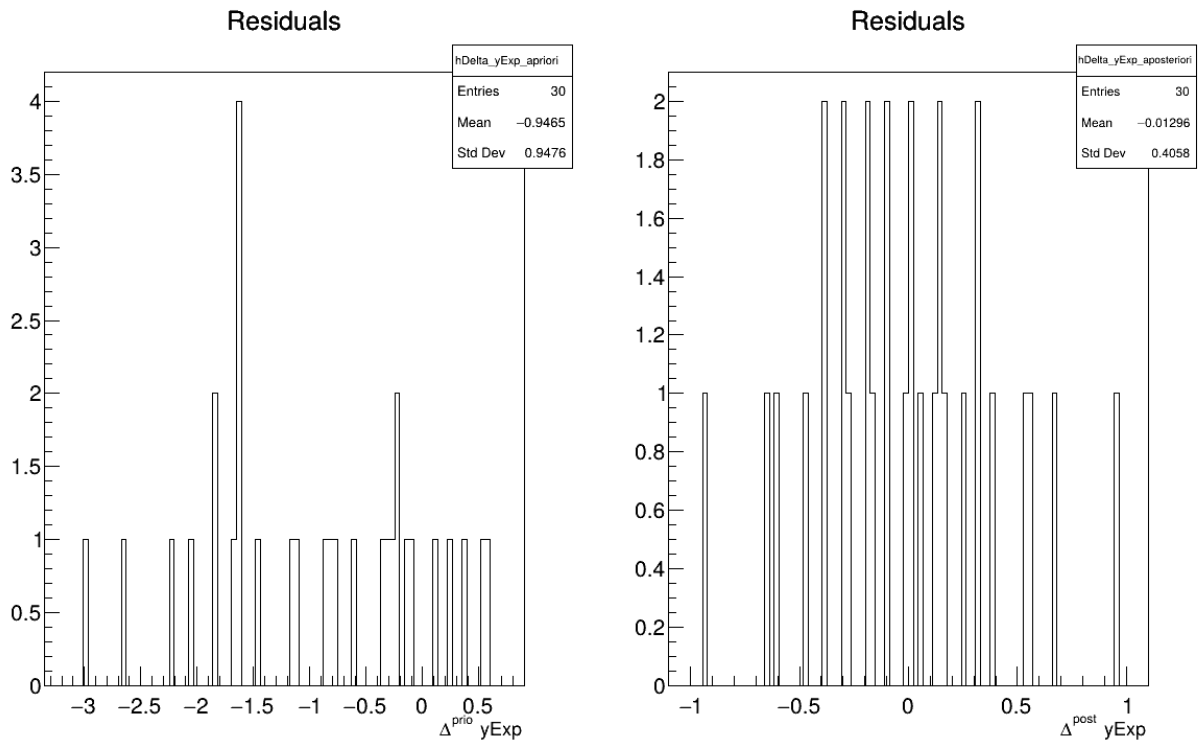


Figure 13.73: Residuals graph of the macro **"calibrationMCMCLinReg.py"**

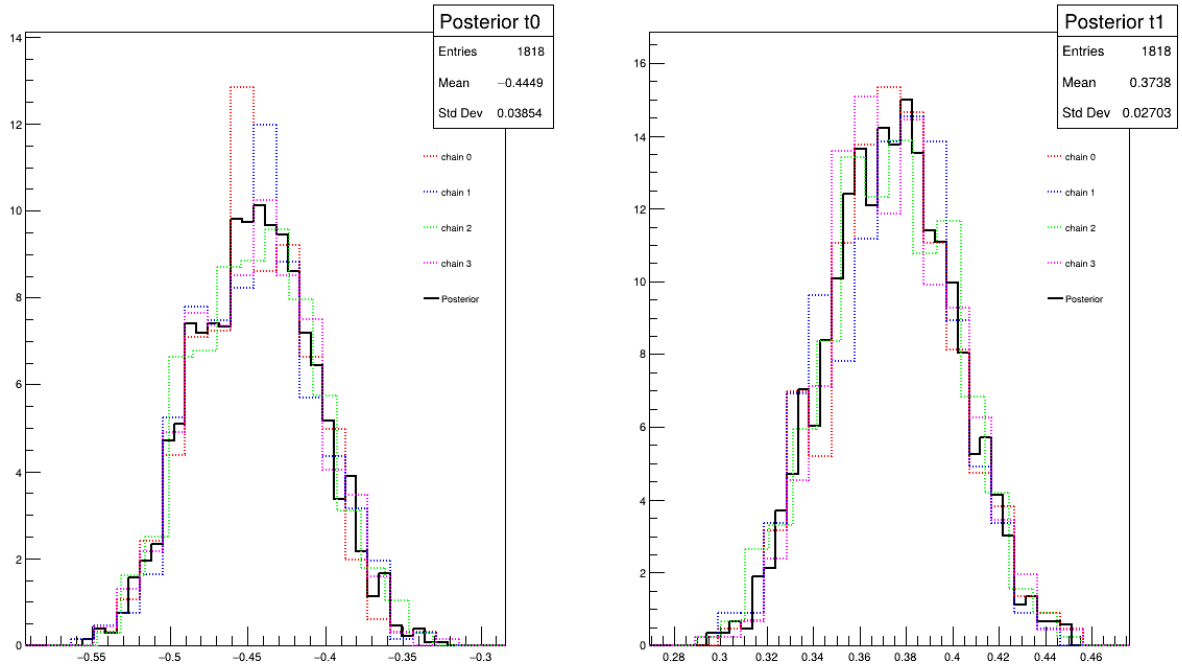


Figure 13.74: Parameter graph of the macro **"calibrationMCMCLinReg.py"**

13.11.7 Macro "calibrationCirce.py"

13.11.7.1 Objective

The goal here is to model the uncertainties and quantify the discrepancy between model outputs and experimental data using the **Circe** method on the dataset `"jdd_circe_summerschool2006_dataserver.dat"`. This dataset contains 150 patterns, each described by four attributes:

- **"exp"**: name of the experimental attribute;
- **"code"**: name of the code output attribute;
- **"sens1"**: name of the attribute corresponding to the derivative of the code output with respect to the first parameter;
- **"sens2"**: name of the attribute corresponding to the derivative of the code output with respect to the second parameter.

```
#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
```

(continues on next page)

(continued from previous page)

```
 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
```

(continues on next page)

(continued from previous page)

```
    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
```

(continues on next page)

(continued from previous page)

```
      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
```

13.11.7.2 Macro Uranie

After loading the reference dataset, the `TCirce` object can be constructed by specifying the reference `TDataServer`, the name of the experimental attribute, the code output attribute, and the derivative attributes. It is then possible to set optional algorithm parameters before launching the process. Finally, the results are extracted and displayed in *Console*.

```
"""
Example of Circe method application
"""
from URANIE import DataServer, Calibration
import ROOT
```

(continues on next page)

(continued from previous page)

```
tds = DataServer.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
# but also Code attribute and sensitivity attributes
tc = Calibration.TCirce(tds, "exp", "code", "sens1,sens2")
# tc.setTolerance(1e-5)
# 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
vBiais = tc.getBVector()
print(" ** vBiais rows["+str(vBiais.GetNrows())+"]")
vBiais.Print()

matC = tc.getCMatrix()
print(" ** matC rows["+str(matC.GetNrows())+"] col ["+str(matC.GetNcols())+"]")
matC.Print()
```

13.11.7.3 Console

```
--- Uranie v4.11/0 --- Developed with ROOT (6.36.06)
                       Copyright (C) 2013-2026 CEA/DES
                       Contact: support-uranie@cea.fr
                       Date: Thu Feb 12, 2026

 *********
 ** 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__
```

(continues on next page)

(continued from previous page)

```
↪        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]

Vector (2)  is as follows

     |        1  |
--------------------
   0 |-0.0131705
   1 |0.0106155


2x2 matrix is as follows

     |      0    |      1    |
-------------------------------
   0 |    0.01612           0
   1 |          0     0.03616

 ** vBiais rows[2]
 ** matC rows[2] col [2]
```

13.12 Macros UncertModeler

13.12.1 Macro "uncertModelerTestsYoungsModulus.py"

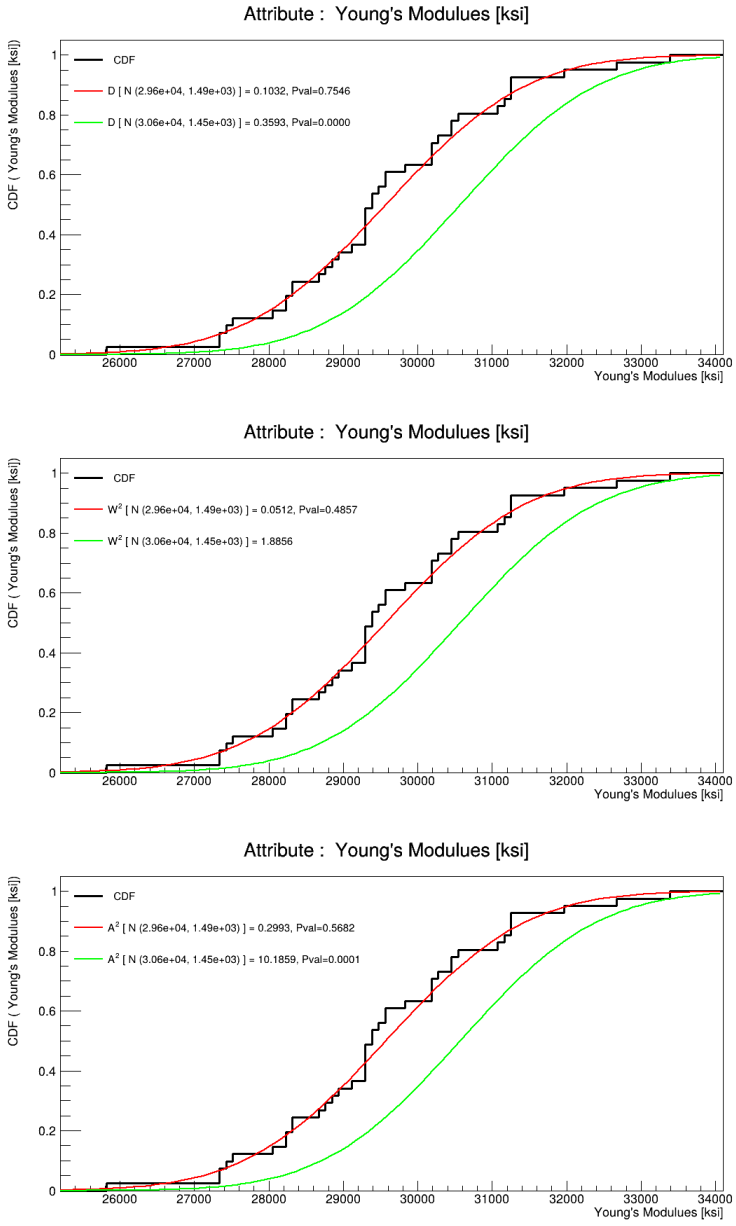
13.12.1.1 Objective

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.

13.12.1.2 Macro Uranie

```
"""
Example of distribution testing with quality criteria on data
"""
from URANIE import DataServer, UncertModeler
import ROOT

tds = DataServer.TDataServer()
tds.fileDataRead("youngsmodulus.dat")

c = ROOT.TCanvas("c1", "Test on youngsmodulus dataset", 13, 38, 1210, 1874)
pad = ROOT.TPad("pad", "pad", 0, 0.03, 1, 1)
pad.Draw()
pad.Divide(1, 3)

tks = UncertModeler.TTestKolmogorovSmirnov(tds, "E")
pad.cd(1)
tks.computeScore("normal:normal(30576,1450)")

tcvm = UncertModeler.TTestCramerVonMises(tds, "E")
pad.cd(2)
tcvm.computeScore("normal:normal(30576,1450)")

tad = UncertModeler.TTestAndersonDarling(tds, "E")
pad.cd(3)
tad.computeScore("normal:normal(30576,1450)")
```

13.12.1.3 Graph



2026-02-13 - Uranie v4.11/0

Figure 13.75: Graph of the macro **"uncertModelerTestsYoungsModulus.py"**

BIBLIOGRAPHY

[App13] W. Appel. *Probabilité pour les non probabilistes*. H & K, Paris, 2013.

[Arn10] G. Arnaud. Manuel d'utilisation de Vizir distribué v2.0. Technical Report, CEA, SFME/LGLS/RT/10-001/A, 2010.

[Bla17] 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.

[BR97] R. Brun and F. Rademakers. ROOT: An object oriented data analysis framework. *Nucl. Instrum. Meth.*, A389:81–86, 1997. doi:10.1016/S0168-9002(97)00048-X.

[FL02] Michael Feathers and B Lepilleur. Cppunit cookbook. 2002.

[FJ05] 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".

[GFB+04] 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*, 97–104. Budapest, Hungary, September 2004.

[GRG+96] Andrew Gelman, Gareth O Roberts, Walter R Gilks, and others. Efficient metropolis jumping rules. *Bayesian statistics*, 5(599-608):42, 1996.

[Gil09] N. Gilardi. Interface python pour la plate-forme uranie. Technical Report, CEA, SFME/LGLS/RT/09-015/A, 2009.

[Hal64] J. H. Halton. Algorithm 247: radical-inverse quasi-random point sequence. *Commun. ACM*, 7(12):701–702, December 1964. URL: http://doi.acm.org/10.1145/355588.365104, doi:10.1145/355588.365104.

[HD02] J. C. Helton and F. J. Davis. Illustration of sampling-based methods for uncertainty and sensitivity analysis. *Risk Analysis*, 22(3):591–622, 2002. URL: http://dx.doi.org/10.1111/0272-4332.00041, doi:10.1111/0272-4332.00041.

[IC82] 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. doi:10.1080/03610918208812265.

[Joh] Steven G. Johnson. The nlopt nonlinear-optimization package. http://ab-initio.mit.edu/nlopt.

[Jol11] Ian Jolliffe. *Principal component analysis*. Springer, 2011.

[JSW98] 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.

[MH07] Ken Martin and Bill Hoffman. An open source approach to developing software in a small organization. *Ieee Software*, 2007.

[MBC00] 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. URL: http://dx.doi.org/10.2307/1271432, doi:10.2307/1271432.

[MOHW07] 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*, June 2007. URL: http://doi.acm.org/10.1145/1236463.1236467, doi:10.1145/1236463.1236467.

[Nvi11] CUDA Nvidia. Nvidia cuda c programming guide. *Nvidia Corporation*, 120(18):8, 2011.

[Pet01] K. Petras. Fast calculation of coefficients in the smolyak algorithm. *Numerical Algorithms*, 26(2):93–109, 2001. URL: http://dx.doi.org/10.1023/A:1016676624575, doi:10.1023/A:1016676624575.

[RGG+97] Gareth O Roberts, Andrew Gelman, Walter R Gilks, and others. Weak convergence and optimal scaling of random walk metropolis algorithms. *The annals of applied probability*, 7(1):110–120, 1997.

[Sobol67] 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. URL: http://www.sciencedirect.com/science/article/pii/0041555367901449, doi:http://dx.doi.org/10.1016/0041-5553(67)90144-9.

[Wil13] 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.

[Wor87] Brian A Worley. Deterministic uncertainty analysis. Technical Report, Oak Ridge National Lab., TN (USA), 1987.