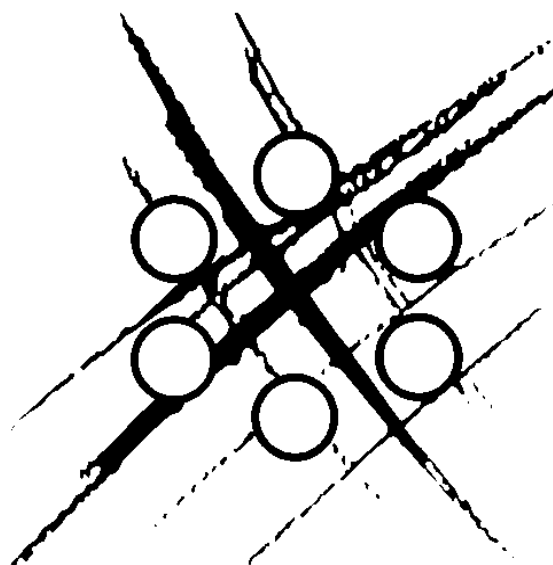




DIRECTION DE L'ÉNERGIE NUCLÉAIRE
DIRECTION DELEGUEE AUX ACTIVITES NUCLEAIRES DE SACLAY
DÉPARTEMENT DE MODÉLISATION DES SYSTÈMES ET STRUCTURES
SERVICE FLUIDES NUMERIQUES, MODELISATION ET ETUDES



RAPPORT DM2S

SFME/LGLS/RT/09-015/A

Interface Python pour la plate-forme URANIE

AUTEURS

N. GILARDI



COMMISSARIAT A L'ÉNERGIE ATOMIQUE



DIRECTION DE L'ÉNERGIE NUCLÉAIRE
DIRECTION DÉLEGUÉE DES ACTIVITÉS NUCLEAIRES
DÉPARTEMENT DE MODÉLISATION DES SYSTÈMES ET STRUCTURES
SERVICE FLUIDES NUMÉRIQUES, MODÉLISATION ET ÉTUDES

RAPPORT DM2S

RÉFÉRENCE : SFME/LGLS/RT/09-015/A

TITRE : Interface Python pour la plate-forme URANIE

AUTEURS	SIGNATURES	AUTEURS	SIGNATURES
N. GILARDI			


Ce rapport présente l'utilisation de l'interface PyROOT pour permettre l'utilisation de la plate-forme URANIE via le langage Python. Grâce à cet outil développé au CERN dans le cadre du projet ROOT, l'accès aux classes d'URANIE ne nécessite aucune intervention de la part des développeurs.

A travers divers exemples, nous présentons les propriétés offertes par PyROOT, ainsi que quelques astuces pour contourner certaines difficultés. Nous terminons par un exemple complet d'appel d'un code externe par URANIE. Une première version est écrite avec la syntaxe C++ de ROOT, une seconde version est écrite avec le langage Python.

MOTS CLES : URANIE, Python, ROOT, PyROOT
AFFAIRE : PIC12
Titre de l'affaire : Développement URANIE - A-PIC12-04


A	27/07/09	15	Visa			
			Nom	F. GAUDIER	V. BERGEAUD	D. CARUGE
			Date			
Indice	Date	Nb.Pages	Vérificateur	Autre visa	Approbateur	

diffusion : normale ☐ restreinte ☐ confidentiel CEA ☐ CD ☐ SD ☐

 DEN/SAC/DM2S SFME/LGLS		SFME/LGLS/RT/09-015/A 27/07/09
	RAPPORT DM2S	Page 3/16
Interface Python pour la plate-forme URANIE		


LISTE DE MODIFICATION

Indice	Date	Motif et description de la modification
A	27/07/2009	Document initial

 DEN/SAC/DM2S SFME/LGLS	<div></div> <div>RAPPORT DM2S</div>	SFME/LGLS/RT/09-015/A 27/07/09 Page 4/16
Interface Python pour la plate-forme URANIE		

SOMMAIRE

1. Introduction.....	4
2. PyROOT.....	5
3. Précautions et recettes.....	5
3.1. Accès aux classes ROOT.....	6
3.2. Accès aux classes URANIE.....	6
3.3. Manipulation des TList et autres séquences de ROOT.....	7
3.4. Pointeurs ou références sur types de base	7
3.5. Gestion de la mémoire.....	8
4. Scripts URANIE.....	8
5. Conclusion.....	13
6. Références.....	14
7. Annexe.....	14

 DEN/SAC/DM2S SFME/LGLS		SFME/LGLS/RT/09-015/A 27/07/09
	RAPPORT DM2S	Page 5/16
Interface Python pour la plate-forme URANIE		

1. Introduction

Grâce à sa simplicité et à sa souplesse, le langage Python [3] a su s'imposer comme l'un des langages interprétés les plus utilisés au monde. En dehors des applications exclusivement programmées dans ce langage, on l'emploie beaucoup comme langage « interface » entre l'utilisateur et des bibliothèques développées dans des langages plus performants mais aussi plus complexes, comme le C, le C++ ou le Fortran.

C'est notamment le cas pour l'environnement ROOT [1]. Développé par le CERN pour traiter les données de physique des hautes énergies, cet outil écrit en C++ propose un grand nombre d'objets destinés à stocker, manipuler et visualiser des données. Toutes ces classes sont accessibles depuis un interpréteur Python grâce à l'interface PyROOT [4].

La plate-forme incertitude URANIE [2], développée au CEA-DEN, se présente comme un module de l'environnement ROOT. De ce fait, l'ensemble de ses fonctionnalités est pris en charge par PyROOT et utilisable via un script Python.

Dans ce document, nous allons présenter rapidement l'interface PyROOT. Nous verrons ensuite quelques points particuliers à prendre en compte lorsque l'on manipule des objets ROOT en Python. Enfin, nous verrons un exemple de script utilisant URANIE pour piloter un code de calcul externe, tantôt en C++, tantôt en Python. Nous verrons également quelques exemples d'affichage disponibles dans URANIE et facilement accessible via PyROOT.

L'objectif de ce document n'est pas de faire un compte rendu exhaustif de l'interface PyROOT, pas plus qu'il n'est un tutoriel pour l'outil URANIE. Le but est avant tout de montrer qu'il est possible (et très simple) d'utiliser URANIE via le langage Python, de signaler quelques « écueils » et de proposer des recettes simples pour les éviter.

On suppose que le lecteur a une connaissance raisonnable tant de Python que du C++. Les exemples concernant URANIE correspondent à des utilisations simples des classes de base de l'outil.


2. PyROOT

Le projet ROOT met à la disposition de ses utilisateurs l'interpréteur CINT, dont la syntaxe est quasi identique au C++. L'idée derrière cet outil est d'accélérer le passage entre la phase de prototypage d'un algorithme et son intégration dans l'outil final.

Lors de la première phase, on se concentre avant tout sur les fonctionnalités, et pour cela, un langage interprété, plus interactif et moins « strict » qu'un langage compilé, est bien adapté. En revanche, la seconde phase implique que le code soit robuste et performant. Dans ce cas, l'utilisation d'un langage compilé est recommandée.

Avec ROOT/CINT, le langage de script et le langage compilé partagent la même syntaxe, ce qui permet de passer d'une phase à l'autre avec un minimum de modification de son code, et de n'apprendre qu'un seul langage de programmation.

Cependant, le C++ possède une syntaxe et des propriétés peu adaptées au prototypage rapide. C'est pour cette raison que de nombreux utilisateurs de ROOT ont souhaité interfacier l'outil avec d'autres langages de scripts, notamment Python et Ruby.

 DEN/SAC/DM2S SFME/LGLS		SFME/LGLS/RT/09-015/A 27/07/09
	RAPPORT DM2S	Page 6/16
Interface Python pour la plate-forme URANIE		

L'interface PyROOT, dans sa version actuelle, a été créée par Wim Lavrijsen, de l'Université de Berkeley (USA). Son objectif est de fournir un accès complet à ROOT depuis un interpréteur Python, et d'accéder à des objets Python depuis l'interpréteur CINT. Ce second point ne semble pas totalement opérationnel et, de toute façon, est de peu d'intérêt dans notre cas. Nous ne nous y intéresserons donc pas dans ce document.

Le point fort de PyROOT est de « construire » l'interfaçage Python de façon totalement transparente pour le développeur ROOT. En effet, toute nouvelle classe intégrée à ROOT et à CINT est automatiquement accessible via PyROOT. Ceci est possible grâce à l'utilisation des dictionnaires de CINT.

Pour accéder aux classes ROOT depuis un script Python, il faut tout d'abord configurer les variables d'environnement adéquates:

- PATH doit contenir \$ROOTSYS/bin
- LD_LIBRARY_PATH doit contenir \$ROOTSYS/lib
- PYTHONPATH doit contenir \$ROOTSYS/lib

Ceci fait, il est alors possible d'importer le module ROOT dans l'interpréteur Python par la commande: `import ROOT`.

Ceci a pour effet de lancer un environnement équivalent à celui de CINT, notamment pour gérer les objets graphiques (TCanvas), et les variables globale (gSystem, gStyle, etc.). A partir de cet instant, tout ce qui était possible via CINT est possible via Python.

3. Précautions et recettes

Que ce soit par choix ou par contrainte, PyROOT introduit un certain nombre de « subtilités » dans l'usage de ROOT via Python.

3.1. ACCÈS AUX CLASSES ROOT


Bien que cela soit déconseillé, il est possible d'importer l'ensemble des classes ROOT par la commande `from ROOT import *`. Toutefois, seuls les noms des variables globales de ROOT seront visibles lors d'une commande `dir()`. Les noms des classes ne sont pas chargés afin de ne pas encombrer l'affichage. Elles sont cependant parfaitement accessibles.

Exemple:

```
>>> from ROOT import *
>>> dir()
['AddressOf', 'MakeNullPointer', 'Template', '__builtins__', '__doc__', '__file__', '__name__',
'gInterpreter', 'gROOT', 'gSystem', 'readline', 'rlcompleter', 'std']
>>> t = TTree()
>>> print t
<ROOT.TTree object at 0x8531418>
```

Si l'on souhaite travailler plus proprement, on peut se contenter de charger le module ROOT et d'accéder à ses composants ainsi:

```
>>> import ROOT
>>> t = ROOT.TTree()
>>> print t
```

 DEN/SAC/DM2S SFME/LGLS		SFME/LGLS/RT/09-015/A 27/07/09
	RAPPORT DM2S	Page 7/16
Interface Python pour la plate-forme URANIE		

```
<ROOT.TTree object at 0x8531418>
```

3.2. ACCÈS AUX CLASSES URANIE

Lorsqu'URANIE est compilé et que l'environnement est configuré comme il faut (lire la documentation d'URANIE [2] pour les détails), les classes d'URANIE sont automatiquement disponibles depuis l'interpréteur CINT. Toutefois, elles sont accessibles via des « espaces de noms ». L'utilisation de ces « namespaces » du C++ dans Python est un peu pénible. En effet, dans le cas d'URANIE, les namespaces ne correspondent pas à de vrais modules Python. Il est donc impossible de les charger comme tel. Pourtant, il est indispensable de les fournir si l'on souhaite que Python puisse créer des objets d'URANIE. Ceci rend l'appel des classes particulièrement verbeux.

Exemple:

```
>>> import ROOT
>>> import ROOT.URANIE.DataServer
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named URANIE.DataServer
>>> t = ROOT.URANIE.DataServer.TDataServer()

--- Uranie v1.1/4 --- Developed with ROOT (5.18/00) by Fabrice Gaudier
                        Copyright (C) 2009 CEA/DEN
                        Version : v1.1/4 - Date : Feb 9 2009
                        All rights reserved, please read http://root.cern.ch/

>>> print t
<ROOT.URANIE::DataServer::TDataServer object ("_tds_") at 0x8232dc8>
```

Une façon de simplifier l'affichage consiste à créer un alias du namespace de la façon suivante:

```
>>> import ROOT
>>> DataServer = ROOT.URANIE.DataServer
>>> t = DataServer.TDataServer()

--- Uranie v1.1/4 --- Developed with ROOT (5.18/00) by Fabrice Gaudier
                        Copyright (C) 2009 CEA/DEN
                        Version : v1.1/4 - Date : Feb 9 2009
                        All rights reserved, please read http://root.cern.ch/


>>> print t
<ROOT.URANIE::DataServer::TDataServer object ("_tds_") at 0x8232d98>
```

3.3. MANIPULATION DES TLIST ET AUTRES SÉQUENCES DE ROOT

La manipulation de données dans ROOT repose beaucoup sur les séquences comme les TList. Ces objets font beaucoup penser aux listes de Python, mais n'en ont pas toutes les propriétés, notamment le « slicing » qui permet de récupérer un sous ensemble d'une liste.

Exemple: on suppose que l'on possède un objet **arbre** de type TTree contenant les branches « b1 », « b2 » et « b3 ».

```
>>> a = arbre.GetListOfBranches()
>>> print a
<ROOT.TObjArray object ("TObjArray") at 0x8397e84>
>>> for branch in a: print branch
...
<ROOT.TBranch object ("b1") at 0x89b5828>
<ROOT.TBranch object ("b2") at 0x89b5d00>
<ROOT.TBranch object ("b3") at 0x89b6220>
>>> a[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

 DEN/SAC/DM2S SFME/LGLS		SFME/LGLS/RT/09-015/A 27/07/09
	RAPPORT DM2S	Page 8/16
Interface Python pour la plate-forme URANIE		

TypeError: none of the 2 overloaded methods succeeded. Full details:

```
TObject*& TObjectArray::operator[](Int_t i) =>
could not convert argument 1 (Objects/longobject.c:223: bad argument to internal function)
TObject* TObjectArray::operator[](Int_t i) =>
could not convert argument 1 (Objects/longobject.c:223: bad argument to internal function)
```

Fort heureusement, en cas de besoin il est très facile de convertir les séquences ROOT en liste Python:

```
>>> b = list(arbre.GetListOfBranches())
>>> print b
[<ROOT.TBranch object ("b1") at 0x89b5828>, <ROOT.TBranch object ("b2") at 0x89b5d00>, <ROOT.TBranch
object ("b3") at 0x89b6220>]
>>> b[:2]
[<ROOT.TBranch object ("b1") at 0x89b5828>, <ROOT.TBranch object ("b2") at 0x89b5d00>]
```

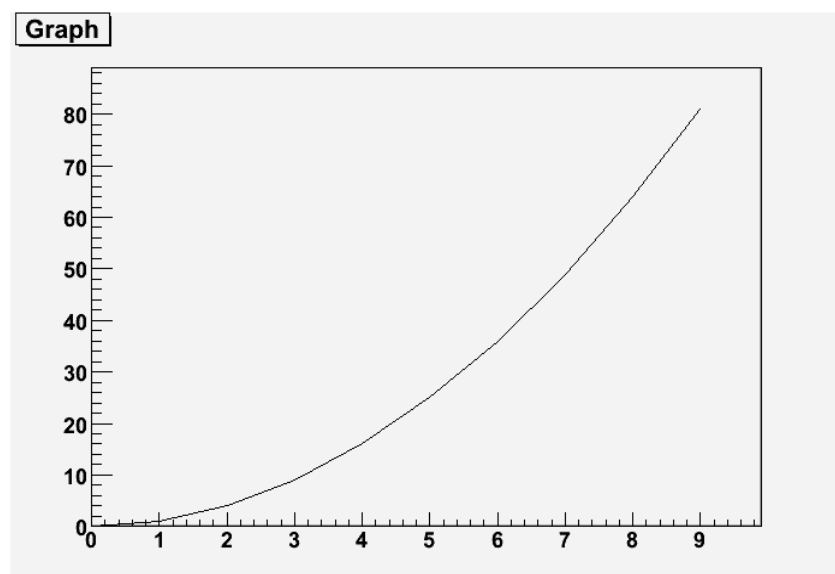
3.4. POINTEURS OU RÉFÉRENCES SUR TYPES DE BASE


La plupart du temps, le passage de paramètres à une fonction ROOT/URANIE est identique en Python et en CINT. Un problème se pose toutefois dans le cas des tableaux et autre pointeurs sur des types de base. En effet, le contenu des listes Python n'est pas typé, ce qui rend la communication avec le C++ un peu délicate.

Heureusement, le module Python « array » permet de créer des objets similaires à des listes mais dont les éléments sont typés. On peut ainsi remplacer l'usage d'un `Double_t*` par une variable définie comme `array.array('d')`.

Exemple: on souhaite créer un objet `TGraph` dont le constructeur prend comme paramètres d'entrée deux tableaux de `Double_t` contenant les coordonnées **x** et **y** des points à afficher:

```
>>> from array import array
>>> x = array('d', range(10))
>>> print x
array('d', [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0])
>>> y = array('d', [a**2 for a in x])
>>> print y
array('d', [0.0, 1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0])
>>> graph = ROOT.TGraph(len(x), x, y)
>>> graph.Draw('AL') # affiche la courbe
```



 DEN/SAC/DM2S SFME/LGLS		SFME/LGLS/RT/09-015/A 27/07/09
	RAPPORT DM2S	Page 9/16
Interface Python pour la plate-forme URANIE		

Au cas où l'on devrait passer une référence sur une variable devant être modifiée par la fonction, il est possible de déclarer une variable avec un type ROOT. Par exemple, si une hypothétique fonction `ROOT::Prod(Double_t a, Double_t b, Double_t &c)` place le produit de **a** par **b** dans la variable **c**, en Python on pourra procéder ainsi:

```
>>> a = 2.0
>>> b = 5.0
>>> c = ROOT.Double()
>>> print c
0.0
>>> ROOT.Prod(a,b,c)
>>> print c
10.0
```

Attention toutefois à ne pas assigner manuellement une valeur numérique à la variable **c**. En effet, celle-ci perdrait alors son type `ROOT.Double()` pour retrouver un type Python.

Un dernier cas de figure est celui où la valeur de retour d'une fonction est un tableau ou un pointeur sur un type de base. Dans ce cas, Python assigne à la variable de retour un pointeur sur un « read-write buffer » et sur lequel les opérations sont extrêmement limitées. Il est donc préférable d'en copier le contenu dans une liste avant de le manipuler. Pour cela, cependant, il faut connaître le nombre d'éléments « utiles » que contient ce buffer.

Exemple: on possède un objet nommé **arbre**, de type `TTree`, contenant les branches « X » et « Y ». On souhaite récupérer les valeurs de « Y » dans une liste Python. La méthode suggérée par les développeurs de ROOT consiste à demander un affichage de « Y ». Cette fonction remplit une variable interne du `TTree` nommée **V1** et retourne le nombre d'éléments de **V1**. Ensuite, la fonction `TTree::GetV1()` retourne un pointeur `Double_t*` sur cette liste de valeurs.

```
>>> nbValues = arbre.Draw('Y','', 'goff') # l'option goff bloque l'affichage
>>> tmp = arbre.GetV1()
>>> print tmp
<Double_t buffer, size 2147483647>
```

A ce stade, nous savons que les **nbValues** premiers éléments de la variable **tmp** sont ceux qui nous intéressent. Nous ne savons rien de ce qu'il y a d'autre dans ce buffer et il est donc plus prudent de ne pas le manipuler directement. Nous allons donc copier les données utiles dans une liste, et ce de la façon suivante:

```
>>> y = [tmp[i] for i in range(nbValues)]
```


La liste **y** contient désormais nos données et nous n'avons plus à nous soucier du buffer **tmp**.

3.5. GESTION DE LA MÉMOIRE

ROOT et Python ayant chacun leur système de « garbage collector », PyROOT doit jongler de l'un à l'autre et, de l'aveu même des développeurs, leur méthode n'est pas infaillible.

N'ayant pas encore été directement impacté par ces problèmes, je n'ai pas d'expérience à transmettre. Je renvoie donc le lecteur à la page du manuel PyROOT traitant (en anglais) du sujet:

<http://wlab.web.cern.ch/wlab/pyroot/memory.html>

 DEN/SAC/DM2S SFME/LGLS		SFME/LGLS/RT/09-015/A 27/07/09
	RAPPORT DM2S	Page 10/16
Interface Python pour la plate-forme URANIE		

4. Scripts URANIE

Voici a présent un exemple de script URANIE faisant appel à un code de calcul externe nommé « flowrate.py ». Ce code, donné en annexe, est indépendant de ROOT et permet de calculer une fonction analytique prenant 8 paramètres en entrée. Cet exemple s'inspire de l'exemple « FlowRate » présenté dans la documentation d'URANIE (accessible en interne CEA):

<file:///home/uranie/tools/html/doc/applications/appliUranieFlowRate.pdf>

Dans un premier temps, nous présentons le script écrit en CINT, le langage de script de ROOT, utilisant la syntaxe du C++. Il peut être lancé au moyen de la commande:

```
root -l testLauncher.C
```

```
{
// Chargement des espaces de noms pour accéder simplement aux classes d'URANIE
using namespace URANIE::DataServer;
using namespace URANIE::Sampler;
using namespace URANIE::Launcher;


// Fichiers de référence
TString referenceDir = gSystem->pwd();
TString inputFileName = referenceDir + TString("/flowrate_input.in");
TString outputFileName = TString("flowrate_output.out");

// Création du DataServer qui va contenir les différentes valeurs des
// paramètres
TDataServer *tds = new TDataServer("tdsflowrate", "DataBase flowrate");

// Création des variables aléatoires qui vont servir à générer les valeurs des
// paramètres du code de calcul FlowRate.
tds->addAttribute(new TUniformDistribution("rw", 0.05, 0.15));
tds->addAttribute(new TUniformDistribution("r", 100.0, 50000.0));
tds->addAttribute(new TUniformDistribution("tu", 63070.0, 115600.0));
tds->addAttribute(new TUniformDistribution("tl", 63.1, 116.0));
tds->addAttribute(new TUniformDistribution("hu", 990.0, 1110.0));
tds->addAttribute(new TUniformDistribution("hl", 700.0, 820.0));
tds->addAttribute(new TUniformDistribution("l", 1120.0, 1680.0));
tds->addAttribute(new TUniformDistribution("kw", 9855.0, 12045.0));

// On associe les attributs à une clef dans le fichier d'entrée du code de
// calcul.
tds->getAttribute("rw")->setKey(inputFileName, "Rw");
tds->getAttribute("r")->setKey(inputFileName, "R");
tds->getAttribute("tu")->setKey(inputFileName, "Tu");
tds->getAttribute("tl")->setKey(inputFileName, "Tl");
tds->getAttribute("hu")->setKey(inputFileName, "Hu");
tds->getAttribute("hl")->setKey(inputFileName, "Hl");
tds->getAttribute("l")->setKey(inputFileName, "L");
tds->getAttribute("kw")->setKey(inputFileName, "Kw");

// Construction d'une liste de valeurs aléatoires pour les entrées du code
// FlowRate, et stockage dans le DataServer
TSampling *sampling = new TSampling(tds, "lhs", 100);
```

 DEN/SAC/DM2S SFME/LGLS		SFME/LGLS/RT/09-015/A 27/07/09
	RAPPORT DM2S	Page 11/16
Interface Python pour la plate-forme URANIE		

```
sampling->generateSample();

// Définition du fichier de sortie
TOutputFile *fout = new TOutputFile(outputFileName);

// Création d'un attribut du DataServer pour récupérer la réponse du code
// FlowRate
TAttribute *tyhat = new TAttribute("yhat");
fout->addAttribute(tyhat);

// Création de l'objet TCode qui va piloter l'appel au code de calcul
TCode *mycode = new TCode(tds, Form("python %s/flowrate.py >> /dev/null",
                                     referenceDir.Data()));

// Ajout des fichiers d'entrée et de sortie à l'objet TCode
mycode->addInputFile( new TInputFile(inputFileName));
mycode->addOutputFile( fout );

// Création de l'objet TLauncher qui va se charger de remplir le fichier
// d'entrée, d'appeler le code de calcul, de récupérer la réponse dans le
// fichier de sortie, et de recommencer pour tous les jeux de paramètres
// contenu dans le DataServer
TLauncher *lanceur = new TLauncher(tds, mycode);
lanceur->setClean();
lanceur->setWorkingDirectory(refDirectory + TString("/tmp/C"));

// Lancement du processus
lanceur->run();

// Exportation des résultats dans un fichier ASCII
tds->exportData("_flowrate_sampler_launcher.dat");
}
```

Le script suivant est identique dans ses fonctionnalités, mais est cette fois écrit en Python. Il peut être lancé avec la commande:


```
python -i testLauncher.py
```

L'option «-i» empêche de quitter l'interpréteur Python à la fin du script, ce qui évite de voir disparaître les graphiques éventuellement générés.

```
# Démarrage de PyROOT et accès aux classes ROOT
import ROOT

# Création d'alias pour les espaces de noms des classes d'URANIE
DataServer = ROOT.URANIE.DataServer
Sampler     = ROOT.URANIE.Sampler
Launcher    = ROOT.URANIE.Launcher

# Détails des paramètres de la fonction FlowRate. Chaque paramètre a un nom
# d'Attribut, une clef, une valeur minimum et une valeur maximum
paramList = [ ("rw","Rw",      0.05,      0.15),
               ("r", "R",      100.0,    50000.0 ),
               ("tu","Tu",    63070.0,  115600.0 ),
               ("tl","Tl",     63.1,     116.0 ) ]
```

 DEN/SAC/DM2S SFME/LGLS		SFME/LGLS/RT/09-015/A 27/07/09
	RAPPORT DM2S	Page 12/16
Interface Python pour la plate-forme URANIE		

```

        ("hu","Hu",    990.0,    1110.0 ),
        ("hl","Hl",    700.0,    820.0 ),
        ("l","L",     1120.0,   1680.0 ),
        ("kw","Kw",   9855.0,  12045.0 )]

# Fichiers de référence
referenceDir  = ROOT.gSystem.pwd()
inputFileName = "flowrate_input.in"
outputFileName = "flowrate_output.out"

# Création du DataServer qui va contenir les différentes valeurs des paramètres
tds = DataServer.TDataServer("tdsflowrate", "DataBase flowrate")

# Création des variables aléatoires qui vont servir à générer les valeurs des
# paramètres du code de calcul FlowRate. On associe également ces variables
# aléatoires à une clef dans le fichier d'entrée du code de calcul.
for p in paramList:
    tds.addAttribute(DataServer.TUniformDistribution(p[0], p[2], p[3]))
    tds.getAttribute(p[0]).setKey(referenceDir + "/" + inputFileName, p[1])

# Construction d'une liste de valeurs aléatoires pour les entrées du code
# FlowRate, et stockage dans le DataServer
sampling = Sampler.TSampling(tds, "lhs", 100)
sampling.generateSample()

# Définition du fichier de sortie
fout = Launcher.TOutputFile(outputFileName)

# Création d'un attribut du DataServer pour récupérer la réponse du code
# FlowRate
tyhat = DataServer.TAttribute("yhat")
fout.addAttribute(tyhat)

# Création de l'objet TCode qui va piloter l'appel au code de calcul
mycode = Launcher.TCode(tds, "python %s/flowrate.py"%referenceDir)


# Ajout des fichiers d'entrée et de sortie à l'objet TCode
mycode.addInputFile(Launcher.TInputFile(referenceDir + "/" + inputFileName))
mycode.addOutputFile( fout )

# Création de l'objet Tlauncher qui va lancer se charger de remplir le fichier
# d'entrée, appeler le code de calcul, récupérer la réponse dans le fichier de
# sortie, et recommencer pour tous les jeux de paramètres contenu dans le DataServer
lanceur = Launcher.TLauncher(tds, mycode)
lanceur.setClean()
lanceur.setWorkingDirectory(referenceDir + "/tmp/py")

# Lancement du processus
lanceur.run()

# Exportation des résultats dans un fichier ASCII
tds.exportData("_flowrate_sampler_launcher_.dat")

```

 DEN/SAC/DM2S SFME/LGLS		SFME/LGLS/RT/09-015/A 27/07/09
	RAPPORT DM2S	Page 13/16
Interface Python pour la plate-forme URANIE		

On observe bien que le passage du code CINT au code Python (et vice versa) est très simple. L'essentiel du travail est en effet effectué via des objets ROOT ou URANIE, et de ce point de vue, la syntaxe est identique entre les deux langages de script.

Il est par exemple très facile de générer des graphiques à partir de PyROOT. Ainsi, si l'on souhaite analyser les résultats précédemment obtenus sous forme graphique, on peut procéder ainsi:

```
# Chargement de PyROOT
import ROOT

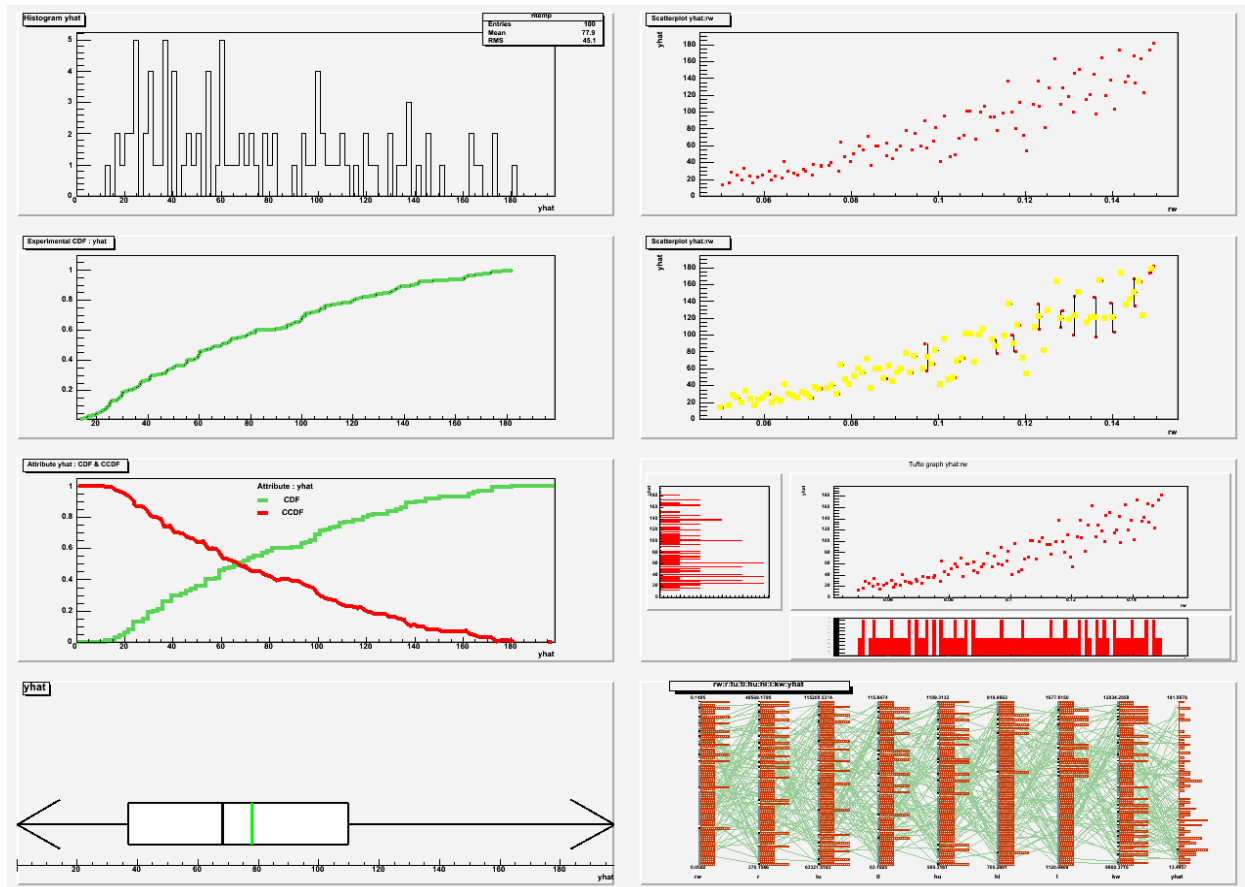
# Construction du DataServer et recuperation des donnees sauvegardees
tds = ROOT.URANIE.DataServer.TDataServer("tdsflowrate","Exemple FlowRate")
tds.fileDataRead("_flowrate_sampler_launcher_.dat")

# Construction d'un canevas pour l'affichage
Canvas = ROOT.TCanvas("c1", "Graph for the Macro loadASCIIFile",5,64,1270,667)

# Subdivision du canevas en 2 colonnes et 4 lignes
Canvas.Divide(2,4)

# Exemples d'affichages
# Histogramme de la variable yhat
Canvas.cd(1); tds.draw("yhat")
# Cumulative expérimentale de la variable yhat
Canvas.cd(3); tds.drawECDF("yhat")
# Histogramme cumulé et son inverse pour la variable yhat
Canvas.cd(5); tds.drawCDF("yhat","", "ccdf")
# Boxplot de la variable yhat
Canvas.cd(7); tds.drawBoxPlot("yhat")
# Scatterplot entre les variables rw et yhat
Canvas.cd(2); tds.draw("yhat:rw")
# Même chose, mais avec des barres d'erreur calculées sur des intervalles en
# abscisse et en ordonnée
Canvas.cd(4); tds.drawProfile("yhat:rw","", "same")
# Scatterplot avec des histogrammes projetés des variables yhat et rw
Canvas.cd(6); tds.drawTufte("yhat:rw")
# Affichage des histogrammes de toutes les variables d'entrées et de leur lien
# avec la variable de sortie
Canvas.cd(8); tds.drawCobWeb("rw:r:tu:tl:hu:hl:l:kw:yhat","", "same")
```

Le résultat de cet affichage est disponible ci-dessous:




Tous ces modes d'affichage sont directement accessibles via un objet de type TDataServer.

En ce qui concerne les performances, on a lancé l'exemple précédent sur une même machine. Le plan d'expérience contenait 1000 points, soit 1000 appels au code externe. Sur trois essais, la meilleure performance du code CINT fut de 40,0 secondes. Egalement sur trois essais, la meilleure performance du code Python fut de 39,8 secondes. Compte tenu de l'imprécision de telles mesures, on peut donc considérer que les performances des deux approches sont identiques.

5. Conclusion

Comme il a été montré dans ce document, l'interfaçage d'URANIE en Python ne nécessite aucun effort de la part des développeurs d'URANIE. Cet énorme avantage en terme de temps de développement et de maintenance a été rendu possible par le choix de baser URANIE sur l'environnement ROOT. Ceci a pour effet de rendre les fonctionnalités d'URANIE automatiquement accessibles via l'interface PyROOT.

De plus, cette capacité d'appeler des fonctionnalités ROOT/URANIE depuis Python est un élément important pour la cohérence de l'environnement logiciel de la DEN. En effet, celui-ci repose sur la plate-forme SALOME pour les aspects environnements de code, et sur URANIE pour la thématique incertitudes/optimisation. Python étant le langage choisi dans SALOME pour l'appel des différents modules, il est possible depuis SALOME d'invoquer des méthodes URANIE via cette interface Python. Cela permettra notamment de lancer des plans d'expérience depuis le superviseur YACS, plans d'expériences ayant été au préalable définis par URANIE. De même, une analyse et une visualisation

 DEN/SAC/DM2S SFME/LGLS		SFME/LGLS/RT/09-015/A 27/07/09
	RAPPORT DM2S	Page 15/16
Interface Python pour la plate-forme URANIE		

ROOT pourra être déclenchée facilement depuis SALOME. Un démonstrateur d'utilisation d'URANIE depuis SALOME pourrait être mis en place en 2010 dans le cadre du projet PICI2.

6. Références

[1]	Site officiel du projet ROOT: http://root.cern.ch
[2]	Manuel Utilisateur URANIE v1 – rapport SFME/LGLS/RT/08-003/A
[3]	Site officiel du langage Python: http://python.org
[4]	Site officiel du projet PyROOT: http://wlv.web.cern.ch/wlv/pyroot/index.html

7. Annexe

Le script suivant est utilisé comme code de calcul externe pour l'exemple « FlowRate »:

```
#!/usr/bin/python

import math

def flowrateModel(parameters):


    rw = parameters[0]
    r = parameters[1]
    tu = parameters[2]
    tl = parameters[3]
    hu = parameters[4]
    hl = parameters[5]
    l = parameters[6]
    kw = parameters[7]

    num = 2.0 * math.pi * tu * (hu - hl)
    lnronrw = math.log(r / rw)
    den = lnronrw * (1.0 + (2.0 * l * tu)/(lnronrw * rw * rw * kw) + tu/tl)

    return (num / den)

if __name__ == "__main__":
    f = open("flowrate_input.in","r")
    paramStrings = f.readlines()
    f.close()

    paramList = []
    for param in ['Rw','R','Tu','Tl','Hu','Hl','L','Kw']:
        for l in paramStrings:
            if param in l:
                tmpLine = l.replace(';','')
                value = float(tmpLine.strip().split('=')[-1])
                paramList.append(value)
                paramStrings.remove(l)
                break
```

 DEN/SAC/DM2S SFME/LGLS		SFME/LGLS/RT/09-015/A 27/07/09
	RAPPORT DM2S	Page 16/16
Interface Python pour la plate-forme URANIE		

```
y = flowrateModel(paramList)

f = open("flowrate_output.out","w")
f.write("yhat = %f ;\n"%y)
f.close()
```

Voici un exemple de fichier « flowrate_input.in »:

```
Rw = 0.0500 ;
R = 33366.67 ;
Tu = 63070.0 ;
Tl = 116.00 ;
Hu = 1110.00 ;
Hl = 768.57;
L = 1200.0 ;
Kw = 11732.14 ;
```

Voici un exemple de fichier « flowrate_output.out »:

```
yhat = 26.180217
```