NISP Toolbox Manual

Michael Baudin (INRIA)
Jean-Marc Martinez (CEA)

Version 0.2
January 2011

Abstract

This document is a brief introduction to the NISP module. We present the installation process of the module in binary from ATOMS or from the sources. We present the configuration functions and the randvar, setrandvar and polychaos classes.

Contents

Chapter 1

Introduction

1.1 The OPUS project

The goal of this toolbox is to provide a tool to manage uncertainties in simulated models. This toolbox is based on the NISP library, where NISP stands for "Non-Intrusive Spectral Projection". This work has been realized in the context of the OPUS project,

http://opus-project.fr

"Open-Source Platform for Uncertainty treatments in Simulation", funded by ANR, the french "Agence Nationale pour la Recherche":

http://www.agence-nationale-recherche.fr

The toolbox is released under the Lesser General Public Licence (LGPL), as all components of the OPUS project.

1.2 The NISP library

The NISP library is based on a set of 3 C++ classes so that it provides an object-oriented framework for uncertainty analysis. The Scilab toolbox provides a pseudo-object oriented interface to this library, so that the two approaches are consistent. The NISP library is release under the LGPL licence.

The NISP library provides three tools, which are detailed below.

- The "randvar" class allows to manage random variables, specified by their distribution law and their parameters. Once a random variable is created, one can generate random numbers from the associated law.
- The "setrandvar" class allows to manage a collection of random variables. This collection is associated with a sampling method, such as MonteCarlo, Sobol, Quadrature, etc... It is possible to build the sample and to get it back so that the experiments can be performed.
- The "polychaos" class allows to manage a polynomial representation of the simulated model. One such object must be associated with a set of experiments which have been performed. This set may be read from a data file. The object is linked with a collection of random

variables. Then the coefficients of the polynomial can be computed by integration (quadrature). Once done, the mean, the variance and the Sobol indices can be directly computed from the coefficients.

The figure 1.1 presents the NISP methodology. The process requires that the user has a numerical solver, which has the form $Y = f(X)$, where X are input uncertain parameters and Y are output random variables. The method is based on the following steps.

- We begin by defining normalized random variables ξ. For example, we may use a random variables in the interval $[0, 1]$ or a Normal random variable with mean 0 and variance 1. This choice allows to define the basis for the polynomial chaos, denoted by $\{\Psi_k\}_{k\geq 0}$. Depending on the type of random variable, the polynomials $\{\Psi_k\}_{k\geq 0}$ are based on Hermite, Legendre or Laguerre polynomials.

- We can now define a Design Of Experiments (DOE) and, with random variable transformations rules, we get the physical uncertain parameters X. Several types of DOE are available: Monte-Carlo, Latin Hypercube Sampling, etc... If N experiments are required, the DOE define the collection of normalized random variables $\{\xi_i\}_{i=1,N}$. Transformation rules allows to compute the uncertain parameters $\{X_i\}_{i=1,N}$, which are the input of the numerical solver f.

- We can now perform the simulations, that is compute the collection of outputs $\{Y_i\}_{i=1,N}$ where $Y_i = f(X_i)$.

- The variables Y are then projected on the polynomial basis and the coefficients y_k are computed by integration or regression.

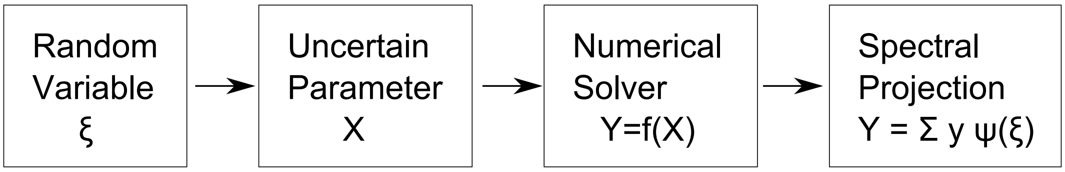

Figure 1.1: The NISP methodology

1.3 The NISP module

The NISP toolbox is available under the following operating systems:

- Linux 32 bits,
- Linux 64 bits,
- Windows 32 bits,
- Mac OS X.

The following list presents the features provided by the NISP toolbox.

- Manage various types of random variables:
  - uniform,
  - normal,
  - exponential,
  - log-normal.
- Generate random numbers from a given random variable,
- Transform an outcome from a given random variable into another,
- Manage various Design of Experiments for sets of random variables,
  - Monte-Carlo,
  - Sobol,
  - Latin Hypercube Sampling,
  - various samplings based on Smolyak designs.
- Manage polynomial chaos expansion and get specific outputs, including
  - mean,
  - variance,
  - quantile,
  - correlation,
  - etc...
- Generate the C source code which computes the output of the polynomial chaos expansion.

This User's Manual completes the online help provided with the toolbox, but does not replace it. The goal of this document is to provide both a global overview of the toolbox and to give some details about its implementation. The detailed calling sequence of each function is provided by the online help and will not be reproduced in this document. The inline help is presented in the figure 1.2.

For example, in order to access to the help associated with the `randvar` class, we type the following statements in the Scilab console.

```
help randvar
```

The previous statements opens the Help Browser and displays the helps page presented in figure

Several demonstration scripts are provided with the toolbox and are presented in the figure 1.4. These demonstrations are available under the "?" question mark in the menu of the Scilab console.

Finally, the unit tests provided with the toolbox cover all the features of the toolbox. When we want to know how to use a particular feature and do not find the information, we can search in the unit tests which often provide the answer.

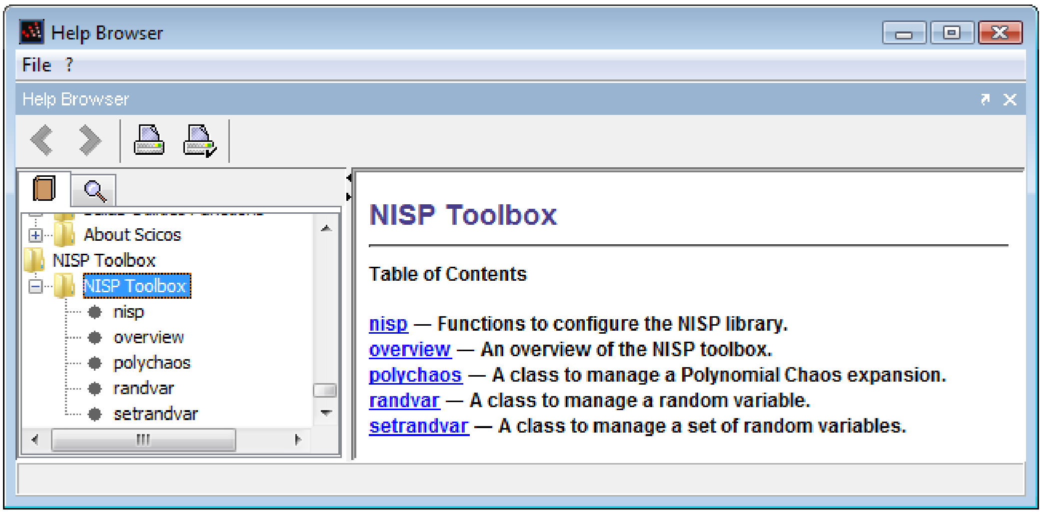

Figure 1.2: The NISP inline help.

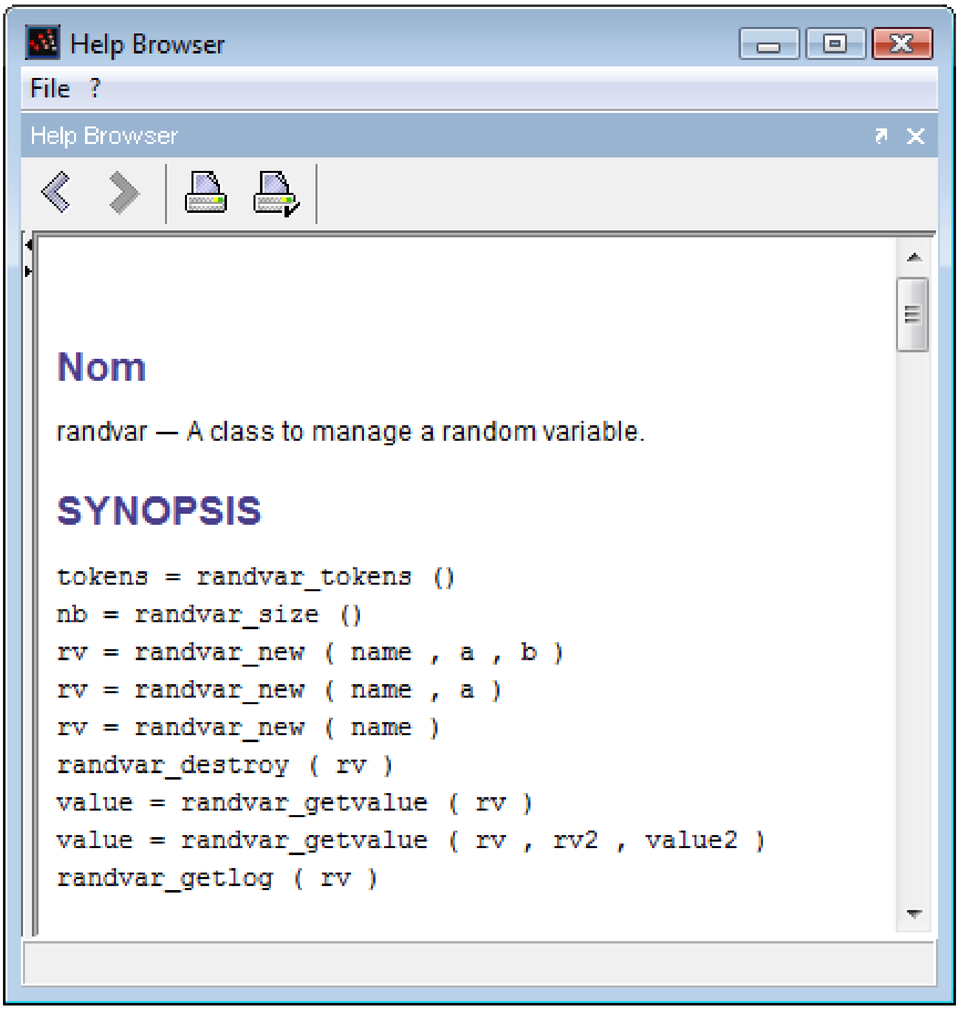

Figure 1.3: The online help of the randvar function.

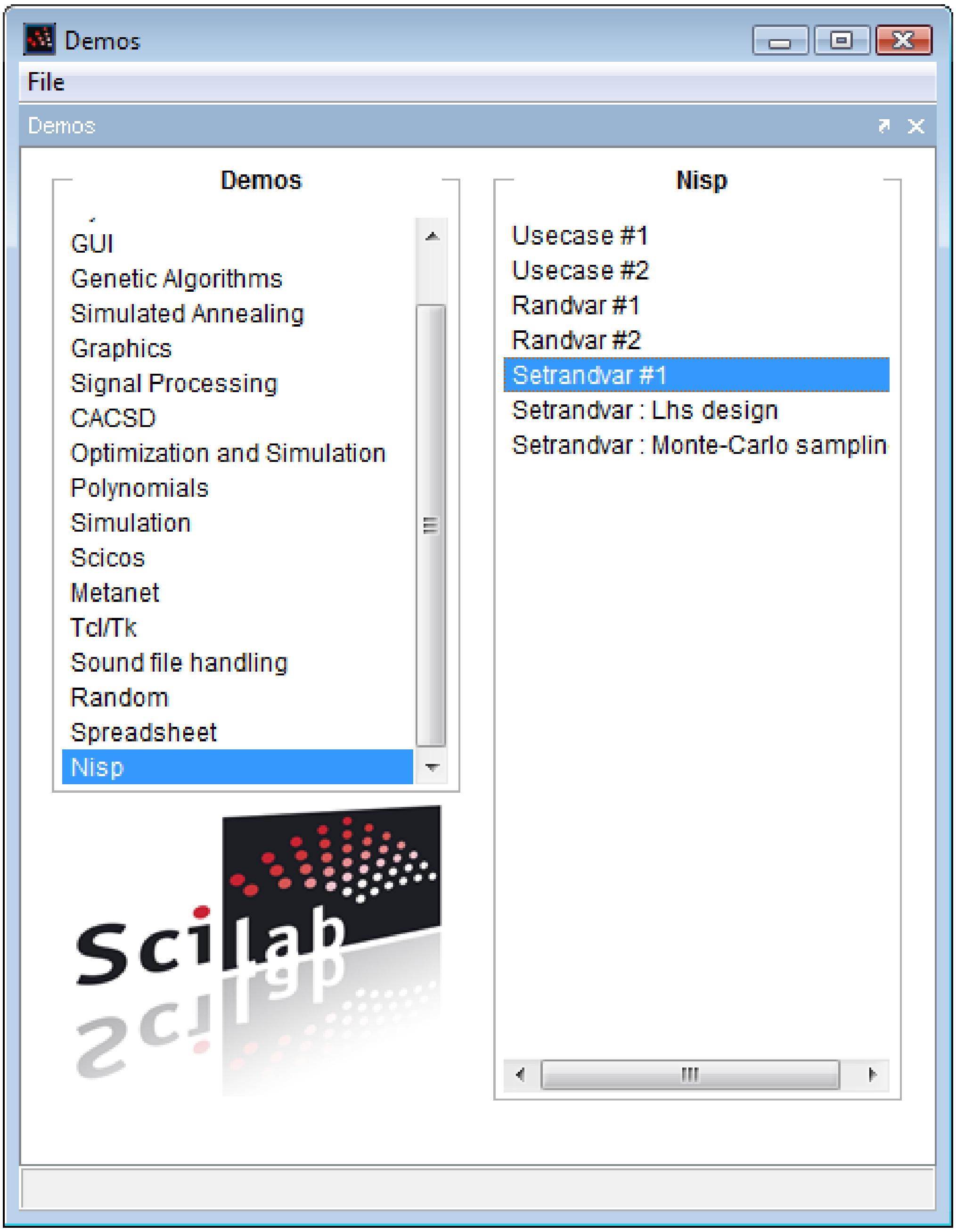

Figure 1.4: Demonstrations provided with the NISP toolbox.

Chapter 2

Installation

In this section, we present the installation process for the toolbox. We present the steps which are required to have a running version of the toolbox and presents the several checks which can be performed before using the toolbox.

2.1 Introduction

There are two possible ways of installing the NISP toolbox in Scilab:

- use the ATOMS system and get a binary version of the toolbox,
- build the toolbox from the sources.

The next two sections present these two ways of using the toolbox.

Before getting into the installation process, let us present some details of the the internal components of the toolbox. The following list is an overview of the content of the directories:

- *tbxnisp/demos* : demonstration scripts
- *tbxnisp/doc* : the documentation
- *tbxnisp/doc/usermanual* : the LaTeXsources of this manual
- *tbxnisp/etc* : startup and shutdow scripts for the toolbox
- *tbxnisp/help* : inline help pages
- *tbxnisp/macros* : Scilab macros files *.sci
- *tbxnisp/sci_gateway* : the sources of the gateway
- *tbxnisp/src* : the sources of the NISP library
- *tbxnisp/tests* : tests
- *tbxnisp/tests/nonreg_tests* : tests after some bug has been identified
- *tbxnisp/tests/unit_tests* : unit tests

The current version is based on the NISP Library v2.1.

2.2 Installing the toolbox from ATOMS

The ATOMS component is the Scilab tool which allows to search, download, install and load toolboxes. ATOMS comes with Scilab v5.2. The Scilab-NISP toolbox has been packaged and is provided mainly by the ATOMS component. The toolbox is provided in binary form, depending on the user's operating system. The Scilab-NISP toolbox is available for the following platforms:

- Windows 32 bits,
- Linux 32 bits, 64 bits,
- Mac OS X.

The ATOMS component allows to use a toolbox based on compiled source code, without having a compiler installed in the system.

Installing the Scilab-NISP toolbox from ATOMS requires the following steps:

- `atomsList()`: prints the list of current toolboxes,
- `atomsShow()`: prints informations about a toolbox,
- `atomsInstall()`: installs a toolbox on the system,
- `atomsLoad()`: loads a toolbox.

Once installed and loaded, the toolbox will be available on the system from session to session, so that there is no need to load the toolbox again: it will be available right from the start of the session.

In the following Scilab session, we use the `atomsList()` function to print the list of all ATOMS toolboxes.

```
--> atomsList()
   ANN_Toolbox - ANN Toolbox
   dde_toolbox - Dynamic Data Exchange client for Scilab
  module_lycee - Scilab pour les lycÃl'es
          NISP - Non Intrusive Spectral Projection
       plotlib - "Matlab-like" Plotting library for Scilab
        scipad - Scipad 7.20
sndfile_toolbox - Read & write sound files
       stixbox - Statistics toolbox for Scilab 5.2
```

In the following Scilab session, we use the `atomsShow()` function to print the details about the NISP toolbox.

```
-->atomsShow("NISP")
      Package : NISP
        Title : NISP
      Summary : Non Intrusive Spectral Projection
      Version : 2.1
       Depend : Category(ies) : Optimization
Maintainer(s) : Pierre Marechal <pierre.marechal@scilab.org>
                Michael Baudin <michael.baudin@scilab.org>
```

```
        Entity : CEA / DIGITEO
       WebSite :       License : LGPL
Scilab Version : >= 5.2.0
        Status : Not installed
   Description : This toolbox allows to approximate a given model,
                 which is associated with input random variables.
                 This toolbox has been created in the context of the
                 OPUS project :
                       http://opus-project.fr/
                 within the workpackage 2.1.1:
                       "Construction de mÃl'ta-modÃÍles"
                 This project has received funding by Agence Nationale
                 de la recherche :
                       http://www.agence-nationale-recherche.fr/
                 See in the help provided in the help/en_US directory
                 of the toolbox for more information about its use.
                 Use cases are presented in the demos directory.
```

In the following Scilab session, we use the `atomsInstall()` function to download and install the binary version of the toolbox corresponding to the current operating system.

```
-->atomsInstall ( "NISP" )
ans =
!NISP  2.1  allusers  D:\Programs\SC3623~1\contrib\NISP\2.1  I  !
```

The `"allusers"` option of the `atomsInstall` function can be used to install the toolbox for all the users of this computer. We finally load the toolbox with the `atomsLoad()` function.

```
-->atomsLoad("NISP")
Start NISP Toolbox
       Load gateways
       Load help
       Load demos
ans =
!NISP  2.1  D:\Programs\SC3623~1\contrib\NISP\2.1  !
```

Now that the toolbox is loaded, it will be automatically loaded at the next Scilab session.

2.3 Installing the toolbox from the sources

In this section, we present the steps which are required in order to install the toolbox from the sources.

In order to install the toolbox from the sources, a compiler is required to be installed on the machine. This toolbox can be used with Scilab v5.1 and Scilab v5.2. We suppose that the archive has been unpacked in the "tbxnisp" directory. The following is a short list of the steps which are required to setup the toolbox.

1. build the toolbox : run the *tbxnisp/builder.sce* script to create the binaries of the library, create the binaries for the gateway, generate the documentation

2. load the toolbox : run the *tbxnisp/load.sce* script to load all commands and setup the documentation

3. setup the startup configuration file of your Scilab system so that the toolbox is known at startup (see below for details),
4. run the unit tests : run the *tbxnisp/runtests.sce* script to perform all unit tests and check that the toolbox is OK
5. run the demos : run the *tbxnisp/rundemos.sce* script to run all demonstration scripts and get a quick interactive overview of its features

The following script presents the messages which are generated when the builder of the toolbox is launched. The builder script performs the following steps:

- compile the NISP C++ library,
- compile the C++ gateway library (the glue between the library and Scilab),
- generate the Java help files from the .xml files,
- generate the loader script.

```
-->exec C:\tbxnisp\builder.sce;
Building sources...
   Generate a loader file
   Generate a Makefile
   Running the Makefile
   Compilation of utils.cpp
   Compilation of blas1_d.cpp
   Compilation of dcdflib.cpp
   Compilation of faure.cpp
   Compilation of halton.cpp
   Compilation of linpack_d.cpp
   Compilation of niederreiter.cpp
   Compilation of reversehalton.cpp
   Compilation of sobol.cpp
   Building shared library (be patient)
   Generate a cleaner file
   Generate a loader file
   Generate a Makefile
   Running the Makefile
   Compilation of nisp_gc.cpp
   Compilation of nisp_gva.cpp
   Compilation of nisp_ind.cpp
   Compilation of nisp_index.cpp
   Compilation of nisp_inv.cpp
   Compilation of nisp_math.cpp
   Compilation of nisp_msg.cpp
   Compilation of nisp_conf.cpp
   Compilation of nisp_ort.cpp
   Compilation of nisp_pc.cpp
   Compilation of nisp_polyrule.cpp
```

```
   Compilation of nisp_qua.cpp
   Compilation of nisp_random.cpp
   Compilation of nisp_smo.cpp
   Compilation of nisp_util.cpp
   Compilation of nisp_va.cpp
   Compilation of nisp_smolyak.cpp
   Building shared library (be patient)
   Generate a cleaner file
Building gateway...
   Generate a gateway file
   Generate a loader file
   Generate a Makefile: Makelib
   Running the makefile
   Compilation of nisp_gettoken.cpp
   Compilation of nisp_gwsupport.cpp
   Compilation of nisp_PolynomialChaos_map.cpp
   Compilation of nisp_RandomVariable_map.cpp
   Compilation of nisp_SetRandomVariable_map.cpp
   Compilation of sci_nisp_startup.cpp
   Compilation of sci_nisp_shutdown.cpp
   Compilation of sci_nisp_verboselevelset.cpp
   Compilation of sci_nisp_verboselevelget.cpp
   Compilation of sci_nisp_initseed.cpp
   Compilation of sci_randvar_new.cpp
   Compilation of sci_randvar_destroy.cpp
   Compilation of sci_randvar_size.cpp
   Compilation of sci_randvar_tokens.cpp
   Compilation of sci_randvar_getlog.cpp
   Compilation of sci_randvar_getvalue.cpp
   Compilation of sci_setrandvar_new.cpp
   Compilation of sci_setrandvar_tokens.cpp
   Compilation of sci_setrandvar_size.cpp
   Compilation of sci_setrandvar_destroy.cpp
   Compilation of sci_setrandvar_freememory.cpp
   Compilation of sci_setrandvar_addrandvar.cpp
   Compilation of sci_setrandvar_getlog.cpp
   Compilation of sci_setrandvar_getdimension.cpp
   Compilation of sci_setrandvar_getsize.cpp
   Compilation of sci_setrandvar_getsample.cpp
   Compilation of sci_setrandvar_setsample.cpp
   Compilation of sci_setrandvar_save.cpp
   Compilation of sci_setrandvar_buildsample.cpp
   Compilation of sci_polychaos_new.cpp
   Compilation of sci_polychaos_destroy.cpp
   Compilation of sci_polychaos_tokens.cpp
   Compilation of sci_polychaos_size.cpp
   Compilation of sci_polychaos_setdegree.cpp
   Compilation of sci_polychaos_getdegree.cpp
   Compilation of sci_polychaos_freememory.cpp
```

```
    Compilation of sci_polychaos_getdimoutput.cpp
    Compilation of sci_polychaos_setdimoutput.cpp
    Compilation of sci_polychaos_getsizetarget.cpp
    Compilation of sci_polychaos_setsizetarget.cpp
    Compilation of sci_polychaos_freememtarget.cpp
    Compilation of sci_polychaos_settarget.cpp
    Compilation of sci_polychaos_gettarget.cpp
    Compilation of sci_polychaos_getdiminput.cpp
    Compilation of sci_polychaos_getdimexp.cpp
    Compilation of sci_polychaos_getlog.cpp
    Compilation of sci_polychaos_computeexp.cpp
    Compilation of sci_polychaos_getmean.cpp
    Compilation of sci_polychaos_getvariance.cpp
    Compilation of sci_polychaos_getcovariance.cpp
    Compilation of sci_polychaos_getcorrelation.cpp
    Compilation of sci_polychaos_getindexfirst.cpp
    Compilation of sci_polychaos_getindextotal.cpp
    Compilation of sci_polychaos_getmultind.cpp
    Compilation of sci_polychaos_getgroupind.cpp
    Compilation of sci_polychaos_setgroupempty.cpp
    Compilation of sci_polychaos_getgroupinter.cpp
    Compilation of sci_polychaos_getinvquantile.cpp
    Compilation of sci_polychaos_buildsample.cpp
    Compilation of sci_polychaos_getoutput.cpp
    Compilation of sci_polychaos_getquantile.cpp
    Compilation of sci_polychaos_getquantwilks.cpp
    Compilation of sci_polychaos_getsample.cpp
    Compilation of sci_polychaos_setgroupaddvar.cpp
    Compilation of sci_polychaos_computeoutput.cpp
    Compilation of sci_polychaos_setinput.cpp
    Compilation of sci_polychaos_propagateinput.cpp
    Compilation of sci_polychaos_getanova.cpp
    Compilation of sci_polychaos_setanova.cpp
    Compilation of sci_polychaos_getanovaord.cpp
    Compilation of sci_polychaos_getanovaordco.cpp
    Compilation of sci_polychaos_realisation.cpp
    Compilation of sci_polychaos_save.cpp
    Compilation of sci_polychaos_generatecode.cpp
    Building shared library (be patient)
    Generate a cleaner file
Generating loader_gateway.sce...
Building help...
Building the master document:
        C:\tbxnisp\help\en_US
Building the manual file [javaHelp] in
C:\tbxnisp\help\en_US.
(Please wait building ... this can take a while)
Generating loader.sce...
```

The following script presents the messages which are generated when the loader of the toolbox is launched. The loader script performs the following steps:

- load the gateway (and the NISP library),
- load the help,
- load the demo.

```
-->exec C:\tbxnisp\loader.sce;
Start NISP Toolbox
        Load gateways
        Load help
        Load demos
```

It is now necessary to setup your Scilab system so that the toolbox is loaded automatically at startup. The way to do this is to configure the Scilab startup configuration file. The directory where this file is located is stored in the Scilab variable `SCIHOME`. In the following Scilab session, we use Scilab v5.2.0-beta-1 in order to know the value of the `SCIHOME` global variable.

```
-->SCIHOME
 SCIHOME  =
 C:\Users\baudin\AppData\Roaming\Scilab\scilab-5.2.0-beta-1
```

On my Linux system, the Scilab 5.1 startup file is located in

`/home/myname/.Scilab/scilab-5.1/.scilab.`

On my Windows system, the Scilab 5.1 startup file is located in

`C:/Users/myname/AppData/Roaming/Scilab/scilab-5.1/.scilab.`

This file is a regular Scilab script which is automatically loaded at Scilab's startup. If that file does not already exist, create it. Copy the following lines into the *.scilab* file and configure the path to the toolboxes, stored in the *SCILABTBX* variable.

```
exec("C:\tbxnisp\loader.sce");
```

The following script presents the messages which are generated when the unit tests script of the toolbox is launched.

```
-->exec C:\tbxnisp\runtests.sce;
Tests beginning the 2009/11/18 at 12:47:45
   TMPDIR = C:\Users\baudin\AppData\Local\Temp\SCI_TMP_6372_
   001/004 - [tbxnisp] nisp..................passed : ref created
   002/004 - [tbxnisp] polychaos1............passed : ref created
   003/004 - [tbxnisp] randvar1..............passed : ref created
   004/004 - [tbxnisp] setrandvar1...........passed : ref created
   ----------------------------------------------------------------
   Summary
   tests                           4 - 100 %
   passed                          0 -   0 %
   failed                          0 -   0 %
   skipped                         0 -   0 %
   length                             3.84 sec
   ----------------------------------------------------------------
Tests ending the 2009/11/18 at 12:47:48\end{verbatim}
```

Chapter 3

Configuration functions

In this section, we present functions which allow to configure the NISP toolbox.

The `nisp_*` functions allows to configure the global behaviour of the toolbox. These functions allows to startup and shutdown the toolbox and initialize the seed of the random number generator. They are presented in the figure 3.1.

`nisp_startup ()`	Starts up the NISP toolbox.
`nisp_shutdown ()`	Shuts down the NISP toolbox.
`level = nisp_verboselevelget ()`	Returns the current verbose level.
`nisp_verboselevelset ( level )`	Sets the value of the verbose level.
`nisp_initseed ( seed )`	Sets the seed of the uniform random number generator.
`nisp_destroyall`	Destroy all current objects.
`nisp_getpath`	Returns the path to the current module.
`nisp_printall`	Prints all current objects.

Figure 3.1: Outline of the configuration methods.

The user has no need to explicitely call the `nisp_startup ()` and `nisp_shutdown ()` functions. Indeed, these functions are called automatically by the *etc/NISP.start* and *etc/NISP.quit* scripts, located in the toolbox directory structure.

The `nisp_initseed ( seed )` is especially useful when we want to have reproductible results. It allows to set the seed of the generator at a particular value, so that the sequence of uniform pseudo-random numbers is deterministic. When the toolbox is started up, the seed is automatically set to 0, which allows to get the same results from session to session.

Chapter 4

The `randvar` class

In this section, we present the `randvar` class, which allows to define a random variable, and to generate random numbers from a given distribution function.

4.1 The distribution functions

In this section, we present the distribution functions provided by the `randvar` class. We especially present the Log-normal distribution function.

4.1.1 Overview

The table 4.1 gives the list of distribution functions which are available with the `randvar` class [3].

Each distribution functions have zero, one or two parameters. One random variable can be specified by giving explicitely its parameters or by using default parameters. The parameters for all distribution function are presented in the figure 4.2, which also presents the conditions which must be satisfied by the parameters.

Name	$f(x)$	$E(X)$	$V(X)$
"Normale"	$\frac{1}{2\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right)$	μ	σ^2
"Uniforme"	$\begin{cases} \frac{1}{b-a}, & \text{if } x \in [a,b[\\ 0 & \text{if } x \notin [a,b[\end{cases}$	$\frac{b+a}{2}$	$\frac{(b-a)^2}{12}$
"Exponentielle"	$\begin{cases} \lambda \exp(-\lambda x), & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$	$\frac{1}{\lambda}$	$\frac{1}{\lambda^2}$
"LogNormale"	$\begin{cases} \frac{1}{\sigma x\sqrt{2\pi}} \exp\left(-\frac{1}{2}\frac{(\ln(x)-\mu)^2}{\sigma^2}\right), & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$	$\exp\left(\mu + \frac{1}{2}\sigma^2\right)$	$\left(\exp(\sigma^2) - 1\right) \exp\left(2\mu + \sigma^2\right)$
"LogUniforme"	$\begin{cases} \frac{1}{x}\frac{1}{\ln(b)-\ln(a)}, & \text{if } x \in [a,b[\\ 0 & \text{if } x \notin [a,b[\end{cases}$	$\frac{b-a}{\ln(b)-\ln(a)}$	$\frac{1}{2}\frac{b^2-a^2}{\ln(b)-\ln(a)} - E(x)$

Figure 4.1: Distributions functions of the `randvar` class. – The expected value is denoted by $E(X)$ and the variance is denoted by $V(X)$.

Name	Parameter #1 : a	Parameter #2 : b	Conditions
"Normale"	$\mu = 0.$	$\sigma = 1.$	$\sigma > 0$
"Uniforme"	$a = 0.$	$b = 1.$	$a < b$
"Exponentielle"	$\lambda = 1.$	-	-
"LogNormale"	$\mu' = 0.1$	$\sigma = 1.0$	$\mu', \sigma > 0$
"LogUniforme"	$a = 0.1$	$b = 1.0$	$a, b > 0,\ a < b$

Figure 4.2: Default parameters for distributions functions.

4.1.2 Parameters of the Log-normal distribution

For the "LogNormale" law, the distribution function is usually defined by the expected value μ and the standard deviation σ of the underlying Normal random variable. But, when we create a LogNormale `randvar`, the parameters to pass to the constructor are the expected value of the LogNormal random variable $E(X)$ and the standard deviation of the underlying Normale random variable σ. The expected value and the variance of the Log Normal law are given by

$$E(X) = \exp\left(\mu + \frac{1}{2}\sigma^2\right) \tag{4.1}$$

$$V(X) = \left(\exp(\sigma^2) - 1\right)\exp\left(2\mu + \sigma^2\right). \tag{4.2}$$

It is possible to invert these formulas, in the situation where the given parameters are the expected value and the variance of the Log Normal random variable. We can invert completely the previous equations and get

$$\mu = \ln(E(X)) - \frac{1}{2}\ln\left(1 + \frac{V(X)}{E(X)^2}\right) \tag{4.3}$$

$$\sigma^2 = \ln\left(1 + \frac{V(X)}{E(X)^2}\right). \tag{4.4}$$

In particular, the expected value μ of with the Normal random variable satisfies the equation

$$\mu = \ln(E(X)) - \sigma^2. \tag{4.5}$$

4.1.3 Uniform random number generation

In this section, we present the generation of uniform random numbers.

The goal of this section is to warn users about a current limitation of the library. Indeed, the random number generator is based on the compiler, so that its quality cannot be guaranted.

The Uniforme law is associated with the parameters $a, b \in \mathbb{R}$ with $a < b$. It produces real values uniform in the interval $[a, b]$.

To compute the uniform random number X in the interval $[a, b]$, a uniform random number in the interval $[0, 1]$ is generated and then scaled with

$$X = a + (b - a)\overline{X}. \tag{4.6}$$

Let us now analyse how the uniform random number $\overline{X} \in [0, 1]$ is computed. The uniform random generator is based on the C function *rand*, which returns an integer n in the interval

$[0, RAND\_MAX[$. The value of the $RAND\_MAX$ variable is defined in the file stdlib.h and is compiler-dependent. For example, with the Visual Studio C++ 2008 compiler, the value is

$$RAND\_MAX = 2^{15} - 1 = 32767. \quad (4.7)$$

A uniform value $\overline{X}$ in the range $[0, 1[$ is computed from

$$\overline{X} = \frac{n}{N}, \quad (4.8)$$

where $N = RAND\_MAX$ and $n \in [0, RAND\_MAX[$.

4.2 Methods

In this section, we give an overview of the methods which are available in the `randvar` class.

4.2.1 Overview

The figure 4.3 presents the methods available in the `randvar` class. The inline help contains the detailed calling sequence for each function and will not be repeated here.

Constructors
`rv = randvar_new ( type [, options])`
Methods
`value = randvar_getvalue ( rv [, options] )`
`randvar_getlog ( rv )`
Destructor
`randvar_destroy ( rv )`
Static methods
`rvlist = randvar_tokens ()`
`nbrv = randvar_size ()`

Figure 4.3: Outline of the methods of the `randvar` class.

4.2.2 The Oriented-Object system

In this section, we present the token system which allows to emulate an oriented-object programming with Scilab. We also present the naming convention we used to create the names of the functions.

The `randvar` class provides the following functions.

- The constructor function `randvar_new` allows to create a new random variable and returns a *token* `rv`.
- The method `randvar_getvalue` takes the token `rv` as its first argument. In fact, all methods takes as their first argument the object on which they apply.

- The destructor `randvar_destroy` allows to delete the current object from the memory of the library.
- The static methods `randvar_tokens` and `randvar_size` allows to quiery the current object which are in use. More specifically, the `randvar_size` function returns the number of current randvar objects and the `randvar_tokens` returns the list of current randvar objects.

In the following Scilab sessions, we present these ideas with practical uses of the toolbox.

Assume that we start Scilab and that the toolbox is automatically loaded. At startup, there are no objects, so that the `randvar_size` function returns 0 and the `randvar_tokens` function returns an empty matrix.

```
-->nb = randvar_size()
 nb  =
    0.
-->tokenmatrix = randvar_tokens()
 tokenmatrix  =
      []
```

We now create 3 new random variables, based on the Uniform distribution function. We store the tokens in the variables `vu1`, `vu2` and `vu3`. These variables are regular Scilab double precision floating point numbers. Each value is a token which represents a random variable stored in the toolbox memory space.

```
-->vu1 = randvar_new("Uniforme")
 vu1  =
    0.
-->vu2 = randvar_new("Uniforme")
 vu2  =
    1.
-->vu3 = randvar_new("Uniforme")
 vu3  =
    2.
```

There are now 3 objects in current use, as indicated by the following statements. The `tokenmatrix` is a row matrix containing regular double precision floating point numbers.

```
-->nb = randvar_size()
 nb  =
    3.
-->tokenmatrix = randvar_tokens()
 tokenmatrix  =
    0.    1.    2.
```

We assume that we have now made our job with the random variables, so that it is time to destroy the random variables. We call the `randvar_destroy` functions, which destroys the variables.

```
-->randvar_destroy(vu1);
-->randvar_destroy(vu2);
-->randvar_destroy(vu3);
```

We can finally check that there are no random variables left in the memory space.

```
-->nb = randvar_size()
 nb  =
    0.
-->tokenmatrix = randvar_tokens()
 tokenmatrix  =
      []
```

Scilab is a wonderful tool to experiment algorithms and make simulations. It happens sometimes that we are managing many variables at the same time and it may happen that, at some point, we are lost. The static methods provides tools to be able to recover from such a situation without closing our Scilab session.

In the following session, we create two random variables.

```
-->vu1 = randvar_new("Uniforme")
 vu1  =
    3.
-->vu2 = randvar_new("Uniforme")
 vu2  =
    4.
```

Assume now that we have lost the token associated with the variable `vu2`. We can easily simulate this situation, by using the `clear`, which destroys a variable from Scilab's memory space.

```
-->clear vu2
-->randvar_getvalue(vu2)
                   !--error 4
Undefined variable: vu2
```

It is now impossible to generate values from the variable `vu2`. Moreover, it may be difficult to know exactly what went wrong and what exact variable is lost. At any time, we can use the `randvar_tokens` function in order to get the list of current variables. Deleting these variables allows to clean the memory space properly, without memory loss.

```
-->randvar_tokens()
 ans  =
    3.    4.
-->randvar_destroy(3)
 ans  =
    3.
-->randvar_destroy(4)
 ans  =
    4.
-->randvar_tokens()
 ans  =
      []
```

4.3 Examples

In this section, we present to examples of use of the `randvar` class. The first example presents the simulation of a Normal random variable and the generation of 1000 random variables. The second example presents the transformation of a Uniform outcome into a LogUniform outcome.

4.3.1 A sample session

We present a sample Scilab session, where the `randvar` class is used to generate samples from the Normale law.

In the following Scilab session, we create a Normale random variable and compute samples from this law. The `nisp_initseed` function is used to initialize the seed for the uniform random variable generator. Then we use the `randvar_new` function to create a new random variable from the Normale law with mean 1. and standard deviation 0.5. The main loop allows to compute 1000 samples from this law, based on calls to the `randvar_getvalue` function. Once the samples are computed, we use the Scilab function `mean` to check that the mean is close to 1 (which is the expected value of the Normale law, when the number of samples is infinite). Finally, we use the `randvar_destroy` function to destroy our random variable. Once done, we plot the empirical distribution function of this sample, with 50 classes.

```
nisp_initseed ( 0 );
mu = 1.0;
sigma = 0.5;
rv = randvar_new("Normale" , mu , sigma);
nbshots = 1000;
values = zeros(nbshots);
for i=1:nbshots
  values(i) = randvar_getvalue(rv);
end
mymean = mean (values);
mysigma = st_deviation(values);
myvariance = variance (values);
mprintf("Mean is : %f (expected = %f)\n", mymean, mu);
mprintf("Standard deviation is : %f (expected = %f)\n", mysigma, sigma);
mprintf("Variance is : %f (expected = %f)\n", myvariance, sigma^2);
randvar_destroy(rv);
histplot(50,values)
xtitle("Histogram of X","X","P(x)")
```

The previous script produces the following output.

```
Mean is : 0.988194 (expected = 1.000000)
Standard deviation is : 0.505186 (expected = 0.500000)
Variance is : 0.255213 (expected = 0.250000)
```

The previous script also produces the figure 4.4.

4.3.2 Variable transformations

In this section, we present the transformation of uniform random variables into other types of variables. The transformations which are available in the `randvar` class are presented in figure 4.5. We begin the analysis by a presentation of the theory required to perform transformations. Then we present some of the many the transformations which are provided by the library.

We now present some additionnal details for the function `randvar_getvalue ( rv , rv2 , value2 )`. This method allows to transform a random variable sample from one law to another. The statement

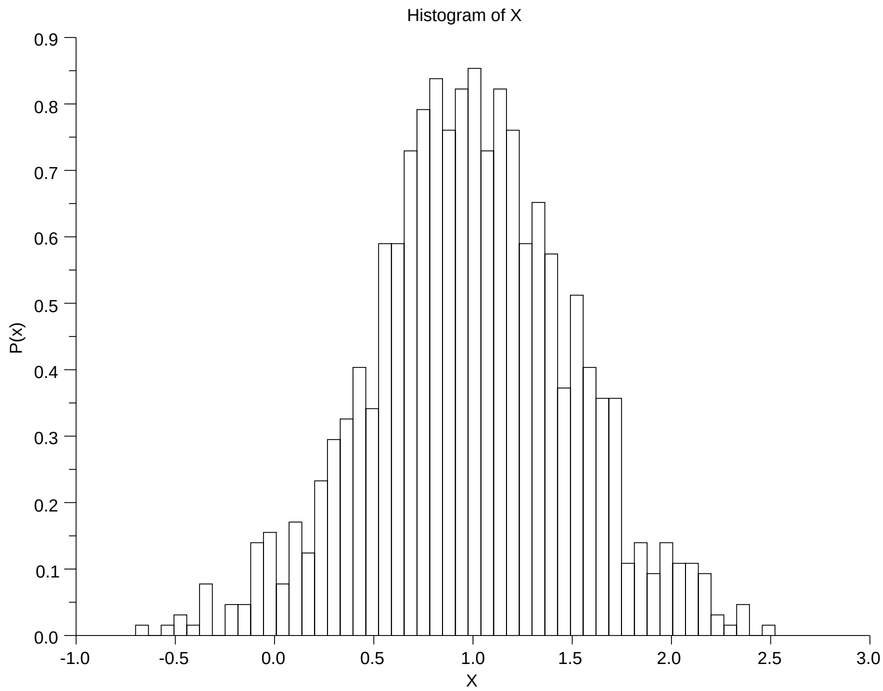

Figure 4.4: The histogram of a Normal random variable with 1000 samples.

Source	**Target**
Normale	
	Normale
	Uniforme
	Exponentielle
	LogNormale
	LogUniforme

Source	**Target**
LogNormale	
	Normale
	Uniforme
	Exponentielle
	LogNormale
	LogUniforme

Source	**Target**
Uniforme	
	Uniforme
	Normale
	Exponentielle
	LogNormale
	LogUniforme

Source	**Target**
LogUniforme	
	Uniforme
	Normale
	Exponentielle
	LogNormale
	LogUniforme

Source	**Target**
Exponentielle	
	Exponentielle

Figure 4.5: Variable transformations available in the `randvar` class.

```
value = randvar_getvalue ( rv , rv2 , value2 )
```

returns a random value from the distribution function of the random variable `rv` by transformation of `value2` from the distribution function of random variable `rv2`.

In the following session, we transform a uniform random variable sample into a LogUniform variable sample. We begin to create a random variable `rv` from a LogUniform law and parameters $a = 10$, $b = 20$. Then we create a second random variable `rv2` from a Uniforme law and parameters $a = 2$, $b = 3$. The main loop is based on the transformation of a sample computed from `rv2` into a sample from `rv`. The `mean` allows to check that the transformed samples have an mean value which corresponds to the random variable `rv`.

```
nisp_initseed ( 0 );
a = 10.0;
b = 20.0;
rv = randvar_new ( "LogUniforme" , a , b );
rv2 = randvar_new ( "Uniforme" , 2 , 3 );
nbshots = 1000;
valuesLou = zeros(nbshots);
for i=1:nbshots
  valuesUni(i) = randvar_getvalue( rv2 );
  valuesLou(i) = randvar_getvalue( rv , rv2 , valuesUni(i) );
end
computed = mean (valuesLou);
mu = (b-a)/(log(b)-log(a));
expected = mu;
mprintf("Expectation=%.5f␣(expected=%.5f)\n",computed,expected);
//
scf();
histplot(50,valuesUni);
xtitle("Empirical␣histogram␣-␣Uniform␣variable","X","P(X)");
scf();
histplot(50,valuesLou);
xtitle("Empirical␣histogram␣-␣Log-Uniform␣variable","X","P(X)");
randvar_destroy(rv);
randvar_destroy(rv2);
```

The previous script produces the following output.

```
Expectation=14.63075 (expected=14.42695)
```

The previous script also produces the figures 4.6 and 4.7.

The transformation depends on the *mother* random variable `rv1` and on the *daughter* random variable `rv`. Specific transformations are provided for all many combinations of the two distribution functions. These transformations will be analysed in the next sections.

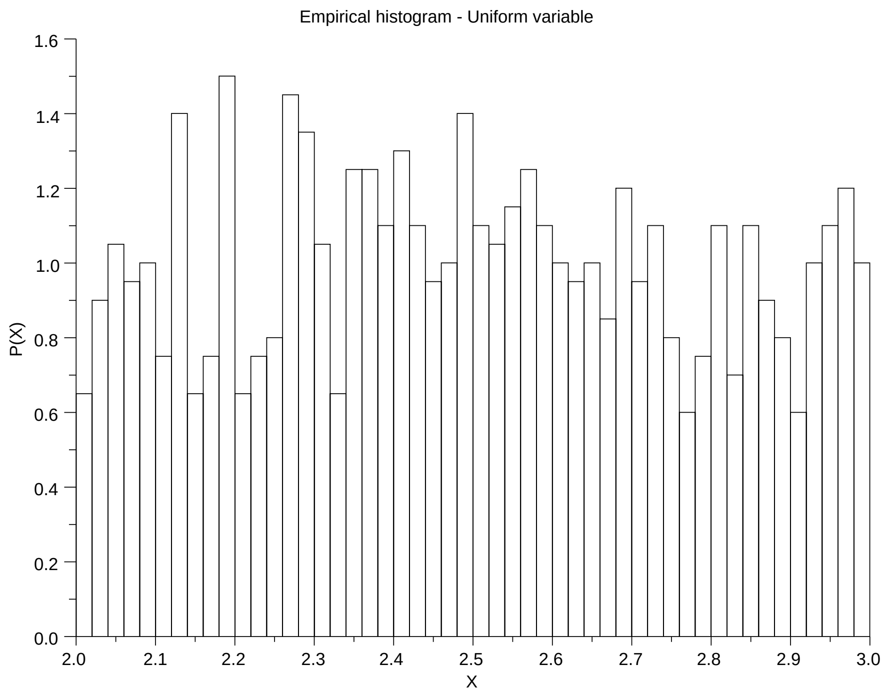

Figure 4.6: The histogram of a Uniform random variable with 1000 samples.

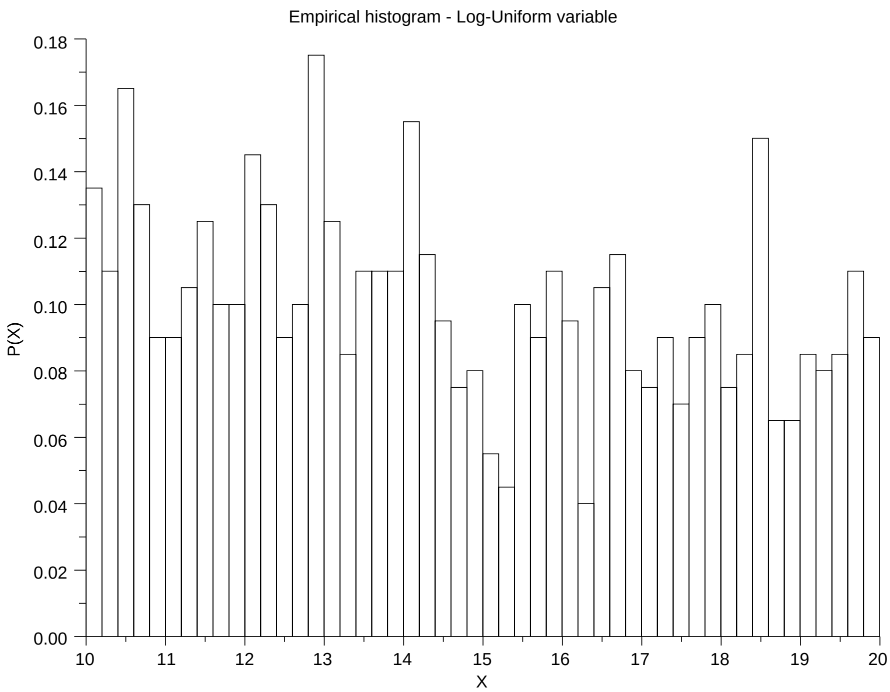

Figure 4.7: The histogram of a Log-Uniform random variable with 1000 samples.

Chapter 5

The `setrandvar` class

In this chapter, we presen the `setrandvar` class. The first section gives a brief outline of the features of this class and the second section present several examples.

5.1 Introduction

The `setrandvar` class allows to manage a collection of random variables and to build a Design Of Experiments (DOE). Several types of DOE are provided:

- Monte-Carlo,
- Latin Hypercube Sampling,
- Smolyak.

Once a DOE is created, we can retrieve the information experiment by experiment or the whole matrix of experiments. This last feature allows to benefit from the fact that Scilab can natively manage matrices, so that we do not have to perform loops to manage the complete DOE. Hence, good performances can be observed, even if the language still is interpreted.

The figure 5.1 presents the methods available in the `setrandvar` class. A complete description of the input and output arguments of each function is available in the inline help and will not be repeated here.

More informations about the Oriented Object system used in this toolbox can be found in the section 4.2.2.

5.2 Examples

In this section, we present examples of use of the `setrandvar` class. In the first example, we present a Scilab session where we create a Latin Hypercube Sampling. In the second part, we present various types of DOE which can be generated with this class.

5.2.1 A Monte-Carlo design with 2 variables

In the following example, we build a Monte-Carlo design of experiments, with 2 input random variables. The first variable is associated with a Normal distribution function and the second

Constructors
`srv = setrandvar_new ( )`
`srv = setrandvar_new ( n )`
`srv = setrandvar_new ( file )`
Methods
`setrandvar_setsample ( srv , name , np )`
`setrandvar_setsample ( srv , k , i , value )`
`setrandvar_setsample ( srv , k , value )`
`setrandvar_setsample ( srv , value )`
`setrandvar_save ( srv , file )`
`np = setrandvar_getsize ( srv )`
`sample = setrandvar_getsample ( srv , k , i )`
`sample = setrandvar_getsample ( srv , k )`
`sample = setrandvar_getsample ( srv )`
`setrandvar_getlog ( srv )`
`nx = setrandvar_getdimension ( srv )`
`setrandvar_freememory ( srv )`
`setrandvar_buildsample ( srv , srv2 )`
`setrandvar_buildsample ( srv , name , np )`
`setrandvar_buildsample ( srv , name , np , ne )`
`setrandvar_addrandvar ( srv , rv )`
Destructor
`setrandvar_destroy ( srv )`
Static methods
`tokenmatrix = setrandvar_tokens ()`
`nb = setrandvar_size ()`

Figure 5.1: Outline of the methods of the `setrandvar` class

variable is associated with a Uniform distribution function. The simulation is based on 1000 experiments.

The function nisp_initseed is used to set the value of the seed to zero, so that the results can be reproduced. The setrandvar_new function is used to create a new set of random variables. Then we create two new random variables with the randvar_new function. These two variables are added to the set with the setrandvar_addrandvar function. The setrandvar_buildsample allows to build the design of experiments, which can be retrieved as matrix with the setrandvar_getsample function. The sampling matrix has np rows and 2 columns (one for each input variable).

```
nisp_initseed(0);
rvu1 = randvar_new("Normale",1,3);
rvu2 = randvar_new("Uniforme",2,3);
//
srvu = setrandvar_new();
setrandvar_addrandvar ( srvu, rvu1);
setrandvar_addrandvar ( srvu, rvu2);
//
np = 5000;
setrandvar_buildsample(srvu, "MonteCarlo",np);
sampling = setrandvar_getsample(srvu);
// Check sampling of random variable #1
mean(sampling(:,1)) // Expectation : 1
// Check sampling of random variable #2
mean(sampling(:,2)) // Expectation : 2.5
//
scf();
histplot(50,sampling(:,1));
xtitle("Empirical histogram of X1");
scf();
histplot(50,sampling(:,2));
xtitle("Empirical histogram of X2");
//
// Clean-up
setrandvar_destroy(srvu);
randvar_destroy(rvu1);
randvar_destroy(rvu2);
```

The previous script produces the following output.

```
-->mean(sampling(:,1)) // Expectation : 1
 ans  =
    1.0064346
-->mean(sampling(:,2)) // Expectation : 2.5
 ans  =
    2.5030984
```

The prevous script also produces the figures 5.2 and 5.3.

We may now want to add the exact distribution to these histograms and compare. The Normal distribution function is not provided by Scilab, but is provided by the Stixbox module. Indeed, the dnorm function of the Stixbox module computes the Normal probability distribution function.

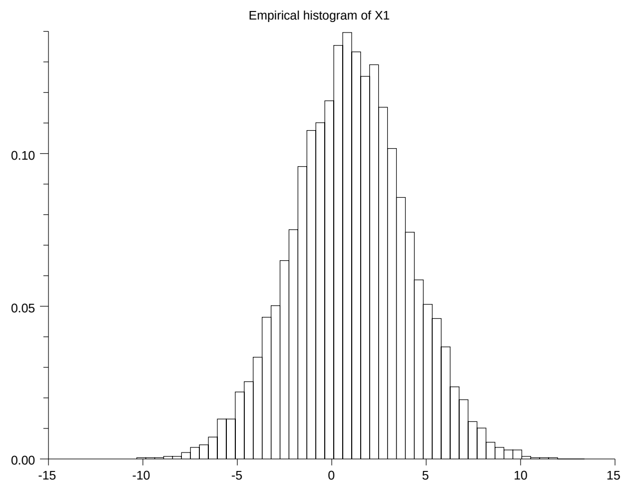

Figure 5.2: Monte-Carlo Sampling - Normal random variable.

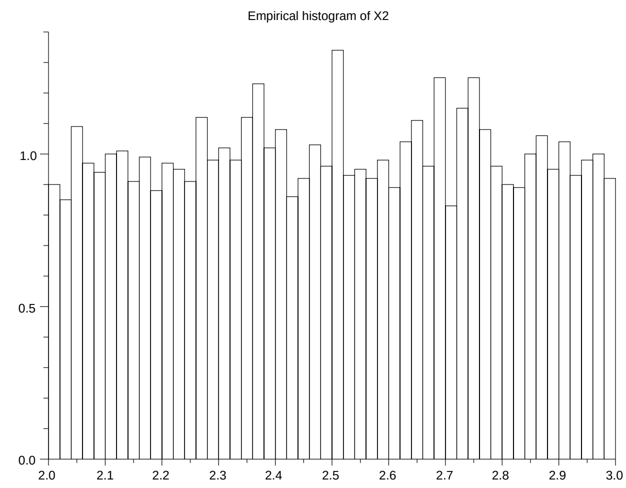

Figure 5.3: Monte-Carlo Sampling - Uniform random variable.

In order to install this module, we can run the atomsInstall function, as in the following script.

```
atomsInstall("stixbox")
```

The following script compares the empirical and theoretical distributions.

```
scf();
histplot(50,sampling(:,1));
xtitle("Empirical␣histogram␣of␣X1");
x=linspace(-15,15,1000);
y = dnorm(x,1,3);
plot(x,y,"r-")
legend(["Empirical","Exact"]);
```

The previous script produces the figure 5.4.

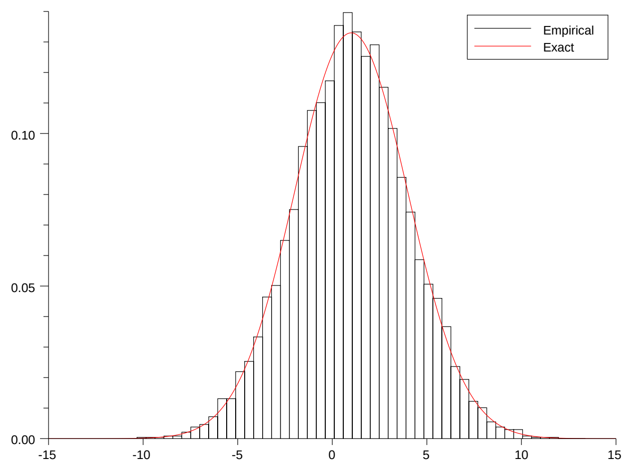

Figure 5.4: Monte-Carlo Sampling - Histogram and exact distribution functions for the first variable.

The following script performs the same comparison for the second variable.

```
scf();
histplot(50,sampling(:,2));
xtitle("Empirical␣histogram␣of␣X2");
x=linspace(2,3,1000);
y=ones(1000,1);
plot(x,y,"r-");
```

The previous script produces the figure 5.5.

5.2.2 A Monte-Carlo design with 2 variables

In this section, we create a Monte-Carlo design with 2 variables.

We are going to use the exponential distribution function, which is not defined in Scilab. The following exppdf function computes the probability distribution function of the exponential distribution function.

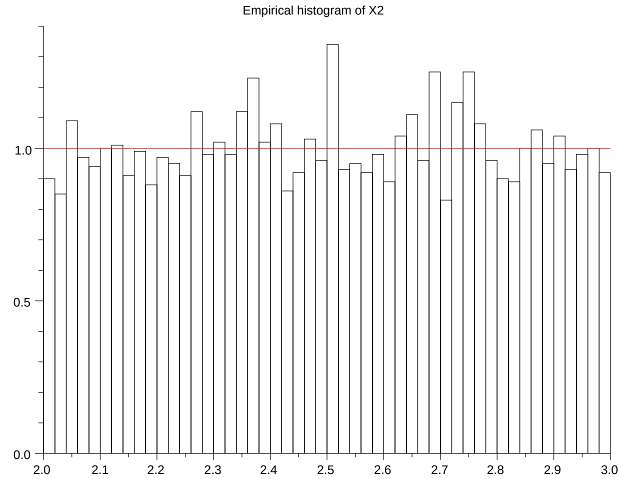

Figure 5.5: Monte-Carlo Sampling - Histogram and exact distribution functions for the second variable.

```
function p = exppdf ( x , lambda )
  p = lambda.*exp(-lambda.*x)
endfunction
```

The following script creates a Monte-Carlo sampling where the first variable is Normal and the second variable is Exponential. Then we compare the empirical histogram and the exact distribution function. We use the `dnorm` function defined in the Stixbox module.

```
nisp_initseed ( 0 );
rv1 = randvar_new("Normale",1.0,0.5);
rv2 = randvar_new("Exponentielle",5.);
// Definition d'un groupe de variables aleatoires
srv = setrandvar_new ( );
setrandvar_addrandvar ( srv , rv1 );
setrandvar_addrandvar ( srv , rv2 );
np = 1000;
setrandvar_buildsample ( srv , "MonteCarlo" , np );
//
sampling = setrandvar_getsample ( srv );
// Check sampling of random variable #1
mean(sampling(:,1)), variance(sampling(:,1))
// Check sampling of random variable #2
min(sampling(:,2)), max(sampling(:,2))
// Plot
scf();
histplot(40, sampling(:,1))
x = linspace(-1,3,1000)';
p = dnorm(x,1,0.5);
plot(x,p,"r-")
```

```
xtitle("Empirical␣histogram␣of␣X1","X","P(X)");
legend(["Empirical","Exact"]);
scf();
histplot(40, sampling(:,2))
x = linspace(0,2,1000)';
p = exppdf ( x , 5 );
plot(x,p,"r-")
xtitle("Empirical␣histogram␣of␣X2","X","P(X)");
legend(["Empirical","Exact"]);
// Clean-up
setrandvar_destroy(srv);
randvar_destroy(rv1);
randvar_destroy(rv2);
```

The previous script produces the figures 5.6 and 5.7.

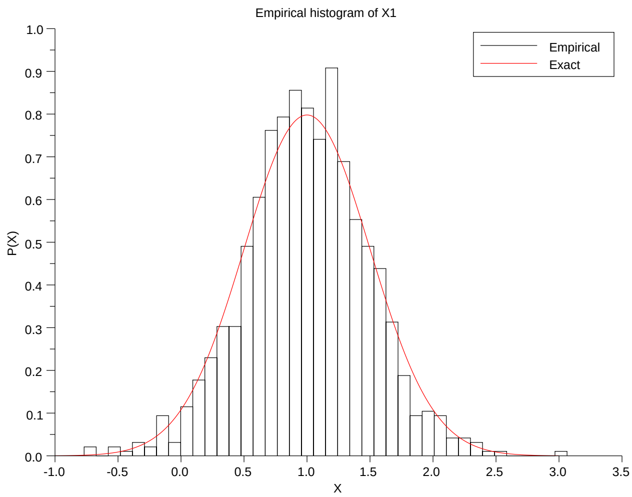

Figure 5.6: Monte-Carlo Sampling - Histogram and exact distribution functions for the first variable.

5.2.3 A LHS design

In this section, we present the creation of a Latin Hypercube Sampling. In our example, the DOE is based on two random variables, the first being Normal with mean 1.0 and standard deviation 0.5 and the second being Uniform in the interval $[2, 3]$.

We begin by defining two random variables with the `randvar_new` function.

```
vu1 = randvar_new("Normale",1.0,0.5);
vu2 = randvar_new("Uniforme",2.0,3.0);
```

Then, we create a collection of random variables with the `setrandvar_new` function which creates here an empty collection of random variables. Then we add the two random variables to the collection.

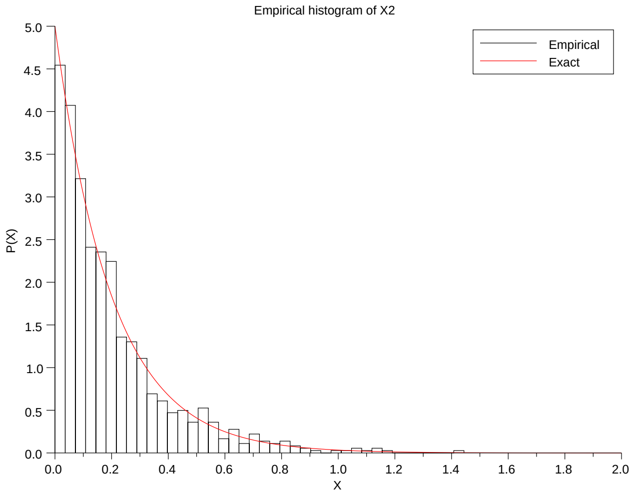

Figure 5.7: Monte-Carlo Sampling - Histogram and exact distribution functions for the second variable.

```
srv = setrandvar_new ( );
setrandvar_addrandvar ( srv , vu1 );
setrandvar_addrandvar ( srv , vu2 );
```

We can now build the DOE so that it is a LHS sampling with 1000 experiments.

```
setrandvar_buildsample ( srv , "Lhs" , 1000 );
```

At this point, the DOE is stored in the memory space of the NISP library, but we do not have a direct access to it. We now call the `setrandvar_getsample` function and store that DOE into the `sampling` matrix.

```
sampling = setrandvar_getsample ( srv );
```

The `sampling` matrix has 1000 rows, corresponding to each experiment, and 2 columns, corresponding to each input random variable.

The following script allows to plot the sampling, which is is presented in figure 5.8.

```
my_handle = scf();
clf(my_handle,"reset");
plot(sampling(:,1),sampling(:,2));
my_handle.children.children.children.line_mode = "off";
my_handle.children.children.children.mark_mode = "on";
my_handle.children.children.children.mark_size = 2;
my_handle.children.title.text = "Latin␣Hypercube␣Sampling";
my_handle.children.x_label.text = "Variable␣#1␣:␣Normale,1.0,0.5";
my_handle.children.y_label.text = "Variable␣#2␣:␣Uniforme,2.0,3.0";
```

The following script allows to plot the histogram of the two variables, which are presented in figures 5.9 and 5.10.

```
// Plot Var #1
```

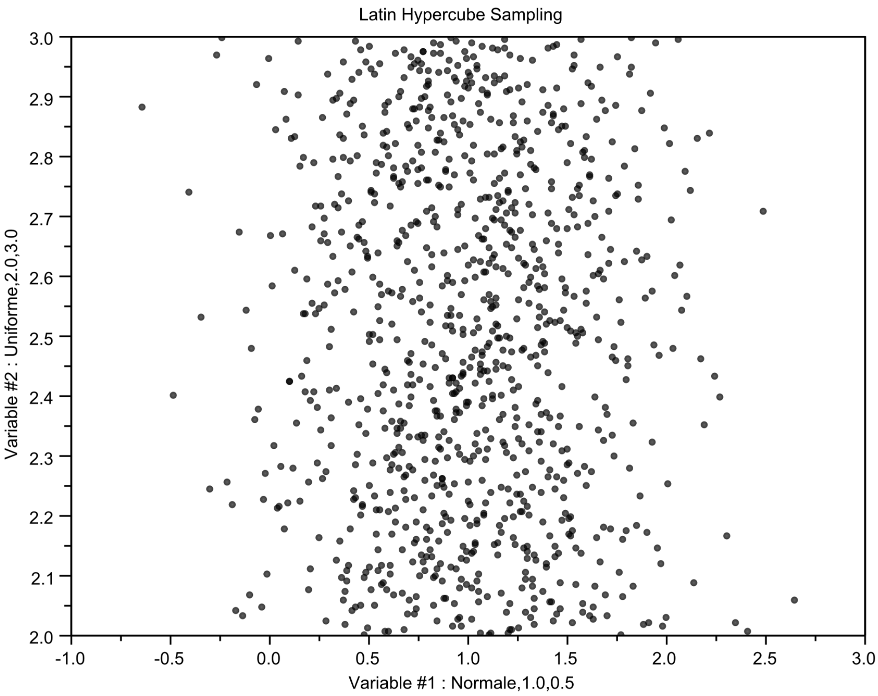

Figure 5.8: Latin Hypercube Sampling - The first variable is Normal, the second variable is Uniform.

```
my_handle = scf();
clf(my_handle,"reset");
histplot ( 50 , sampling(:,1))
my_handle.children.title.text = "Variable␣#1␣:␣Normale,1.0,0.5";
// Plot Var #2
my_handle = scf();
clf(my_handle,"reset");
histplot ( 50 , sampling(:,2))
my_handle.children.title.text = "Variable␣#2␣:␣Uniforme,2.0,3.0";
```

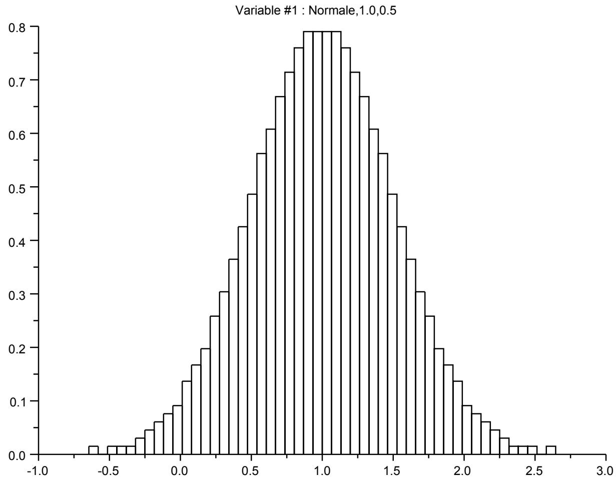

Figure 5.9: Latin Hypercube Sampling - Normal random variable.

We can use the `mean` and `variance` on each random variable and check that the expected result is computed. We insist on the fact that the `mean` and `variance` functions are not provided by the NISP library: these are pre-defined functions which are available in the Scilab library. That means that any Scilab function can be now used to process the data generated by the toolbox.

```
for ivar = 1:2
  m = mean(sampling(:,ivar))
  mprintf("Variable␣#%d,␣Mean␣:␣%f\n",ivar,m)
  v = variance(sampling(:,ivar))
  mprintf("Variable␣#%d,␣Variance␣:␣%f\n",ivar,v)
end
```

The previous script produces the following output.

```
Variable #1, Mean : 1.000000
Variable #1, Variance : 0.249925
Variable #2, Mean : 2.500000
Variable #2, Variance : 0.083417
```

Our numerical simulation is now finished, but we must destroy the objects so that the memory managed by the toolbox is deleted.

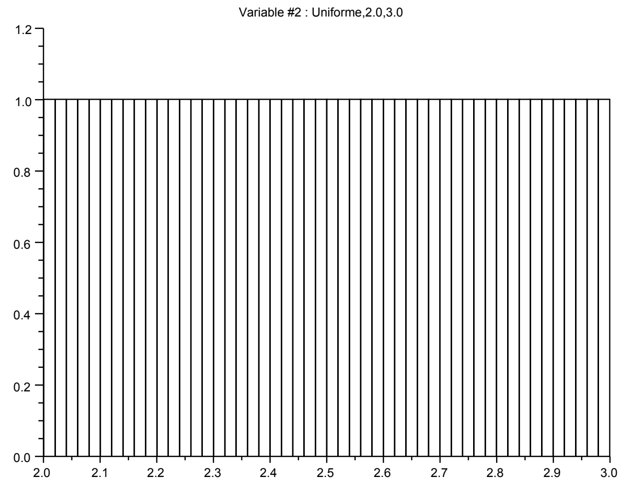

Figure 5.10: Latin Hypercube Sampling - Uniform random variable.

```
randvar_destroy(vu1)
randvar_destroy(vu2)
setrandvar_destroy(srv)
```

5.2.4 Other types of DOEs

The following Scilab session allows to generate a Monte-Carlo sampling with two uniform variables in the interval $[-1, 1]$. The figure 5.11 presents this sampling and the figures 5.12 and 5.13 present the histograms of the two uniform random variables.

```
vu1 = randvar_new("Uniforme",-1.0,1.0);
vu2 = randvar_new("Uniforme",-1.0,1.0);
srv = setrandvar_new ( );
setrandvar_addrandvar ( srv , vu1 );
setrandvar_addrandvar ( srv , vu2 );
setrandvar_buildsample ( srv , "MonteCarlo" , 1000 );
sampling = setrandvar_getsample ( srv );
randvar_destroy(vu1);
randvar_destroy(vu2);
setrandvar_destroy(srv);
```

It is easy to change the type of sampling by modifying the second argument of the `setrandvar_buildsa` function. This way, we can create the Petras, Quadrature and Sobol sampling presented in figures 5.14, 5.15 and 5.16.

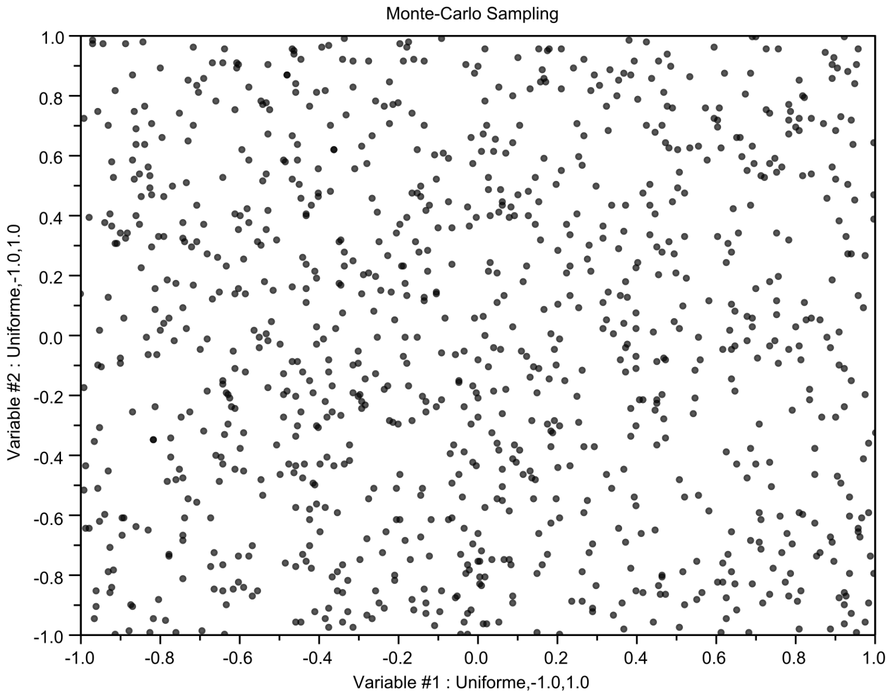

Figure 5.11: Monte-Carlo Sampling - Two uniform variables in the interval $[-1, 1]$.

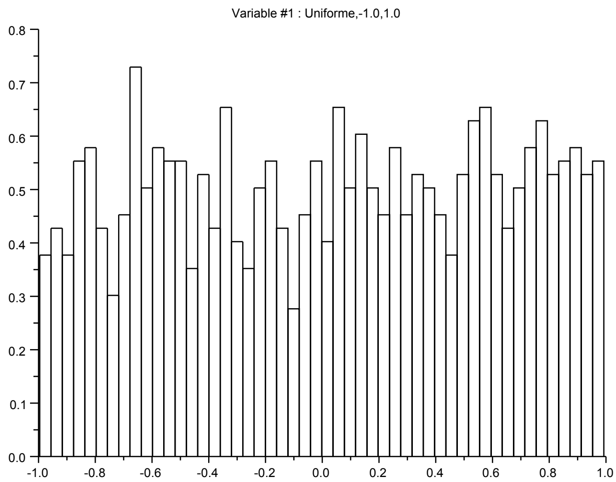

Figure 5.12: Latin Hypercube Sampling - First uniform variable in $[-1, 1]$.

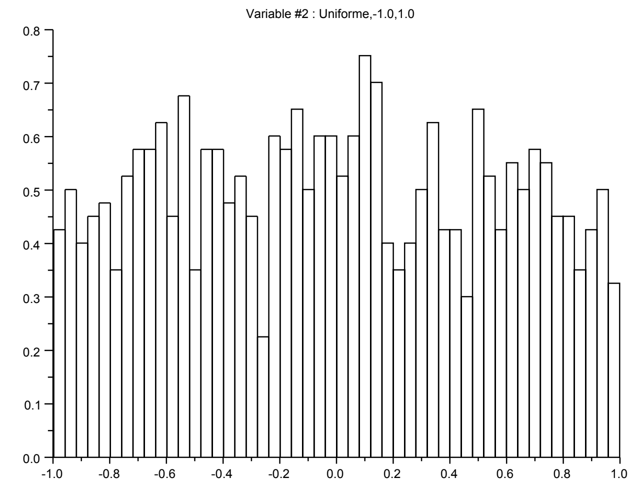

Figure 5.13: Latin Hypercube Sampling - Second uniform variable in $[-1, 1]$.

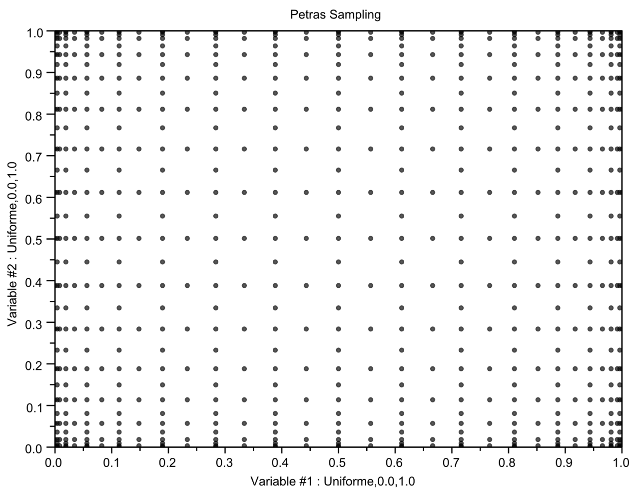

Figure 5.14: Petras sampling - Two uniform variables in the interval $[-1, 1]$.

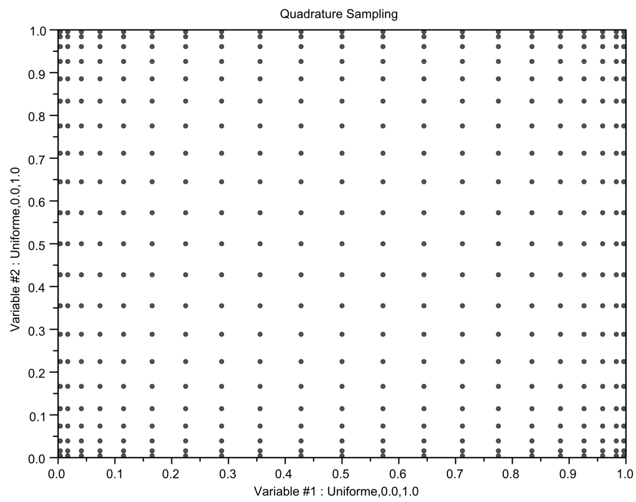

Figure 5.15: Quadrature sampling - Two uniform variables in the interval $[-1, 1]$.

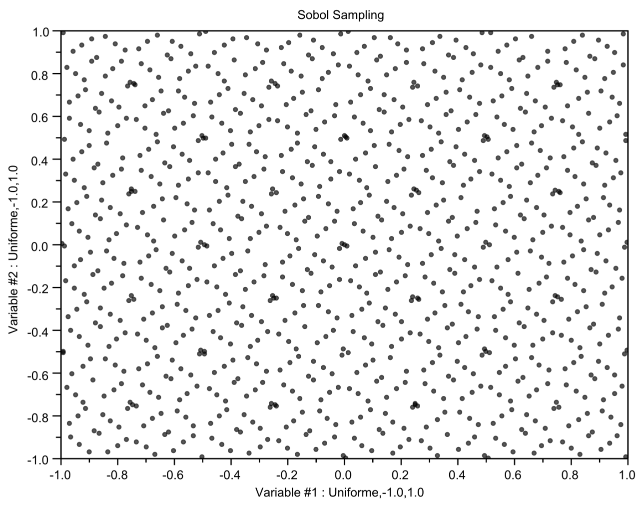

Figure 5.16: Sobol sampling - Two uniform variables in the interval $[-1, 1]$.

Chapter 6

The `polychaos` class

6.1 Introduction

The `polychaos` class allows to manage a polynomial chaos expansion. The coefficients of the expansion are computed based on given numerical experiments which creates the association between the inputs and the outputs. Once computed, the expansion can be used as a regular function. The mean, standard deviation or quantile can also be directly retrieved.

The tool allows to get the following results:

- mean,
- variance,
- quantile,
- correlation, etc...

Moreover, we can generate the C source code which computes the output of the polynomial chaos expansion. This C source code is stand-alone, that is, it is independent of both the NISP library and Scilab. It can be used as a meta-model.

The figure 6.1 presents the most commonly used methods available in the `polychaos` class. More methods are presented in figure 6.2. The inline help contains the detailed calling sequence for each function and will not be repeated here. More than 50 methods are available and most of them will not be presented here.

More informations about the Oriented Object system used in this toolbox can be found in the section 4.2.2.

6.2 Examples

In this section, we present to examples of use of the `polychaos` class.

6.2.1 Product of two random variables

In this section, we present the polynomial expansion of the product of two random variables. We analyse the Scilab script and present the methods which are available to perform the sensi-

Constructors
`pc = polychaos_new ( file )`
`pc = polychaos_new ( srv , ny )`
`pc = polychaos_new ( pc , nopt , varopt )`
Methods
`polychaos_setsizetarget ( pc , np )`
`polychaos_settarget ( pc , output )`
`polychaos_setinput ( pc , invalue )`
`polychaos_setdimoutput ( pc , ny )`
`polychaos_setdegree ( pc , no )`
`polychaos_getvariance ( pc )`
`polychaos_getmean ( pc )`
Destructor
`polychaos_destroy (pc)`
Static methods
`tokenmatrix = polychaos_tokens ()`
`nb = polychaos_size ()`

Figure 6.1: Outline of the methods of the `polychaos` class

tivity analysis. This script is based on the NISP methodology, which has been presented in the Introduction chapter. We will use the figure 1.1 as a framework and will follow the steps in order.

In the following Scilab script, we define the function `Example` which takes a vector of size 2 as input and returns a scalar as output.

```
function y = Exemple (x)
  y(:,1) = x(:,1) .* x(:,2)
endfunction
```

We now create a collection of two stochastic (normalized) random variables. Since the random variables are normalized, we use the default parameters of the `randvar_new` function. The normalized collection is stored in the variable `srvx`.

```
vx1 = randvar_new("Normale");
vx2 = randvar_new("Uniforme");
srvx = setrandvar_new();
setrandvar_addrandvar ( srvx , vx1 );
setrandvar_addrandvar ( srvx , vx2 );
```

We create a collection of two uncertain parameters. We explicitely set the parameters of each random variable, that is, the first Normal variable is associated with a mean equal to 1.0 and a standard deviation equal to 0.5, while the second Uniform variable is in the interval $[1.0, 2.5]$. This collection is stored in the variable `srvu`.

```
vu1 = randvar_new("Normale",1.0,0.5);
vu2 = randvar_new("Uniforme",1.0,2.5);
srvu = setrandvar_new();
setrandvar_addrandvar ( srvu , vu1 );
setrandvar_addrandvar ( srvu , vu2 );
```

Methods

```
output = polychaos_gettarget ( pc )
np = polychaos_getsizetarget ( pc )
polychaos_getsample ( pc , k , ovar )
polychaos_getquantile ( pc , k )
polychaos_getsample ( pc )
polychaos_getquantile ( pc , alpha )
polychaos_getoutput ( pc )
polychaos_getmultind ( pc )
polychaos_getlog ( pc )
polychaos_getinvquantile ( pc , threshold )
polychaos_getindextotal ( pc )
polychaos_getindexfirst ( pc )
ny = polychaos_getdimoutput ( pc )
nx = polychaos_getdiminput ( pc )
p = polychaos_getdimexp ( pc )
no = polychaos_getdegree ( pc )
polychaos_getcovariance ( pc )
polychaos_getcorrelation ( pc )
polychaos_getanova ( pc )
polychaos_generatecode ( pc , filename , funname )
polychaos_computeoutput ( pc )
polychaos_computeexp ( pc , srv , method )
polychaos_computeexp ( pc , pc2 , invalue , varopt )
polychaos_buildsample ( pc , type , np , order )
```

Figure 6.2: More methods from the `polychaos` class

The first design of experiment is build on the stochastic set `srvx` and based on a Quadrature type of DOE. Then this DOE is transformed into a DOE for the uncertain collection of parameters `srvu`.

```
degre = 2;
setrandvar_buildsample ( srvx , "Quadrature" , degre );
setrandvar_buildsample ( srvu , srvx );
```

The next steps will be to create the polynomial and actually perform the DOE. But before doing this, we can take a look at the DOE associated with the stochastic and uncertain collection of random variables. We can use the `setrandvar_getsample` twice and get the following output.

```
-->setrandvar_getsample(srvx)
 ans  =

  - 1.7320508    0.1127017
  - 1.7320508    0.5
  - 1.7320508    0.8872983
    0.           0.1127017
    0.           0.5
    0.           0.8872983
    1.7320508    0.1127017
    1.7320508    0.5
    1.7320508    0.8872983
-->setrandvar_getsample(srvu)
 ans  =

    0.1339746    1.1690525
    0.1339746    1.75
    0.1339746    2.3309475
    1.           1.1690525
    1.           1.75
    1.           2.3309475
    1.8660254    1.1690525
    1.8660254    1.75
    1.8660254    2.3309475
```

These two matrices are a 9×2 matrices, where each line represents an experiment and each column represents an input random variable. The stochastic (normalized) `srvx` DOE has been created first, then the `srvu` has been deduced from `srvx` based on random variable transformations.

We now use the `polychaos_new` function and create a new polynomial `pc`. The number of input variables corresponds to the number of variables in the stochastic collection `srvx`, that is 2, and the number of output variables is given as the input argument `ny`. In this particular case, the number of experiments to perform is equal to `np=9`, as returned by the `setrandvar_getsize` function. This parameter is passed to the polynomial `pc` with the `polychaos_setsizetarget` function.

```
ny = 1;
pc = polychaos_new ( srvx , ny );
np = setrandvar_getsize(srvx);
polychaos_setsizetarget(pc,np);
```

In the next step, we perform the simulations prescribed by the DOE. We perform this loop in the Scilab language and make a loop over the index `k`, which represents the index of the current experiment, while `np` is the total number of experiments to perform. For each loop, we get the input from the uncertain collection `srvu` with the `setrandvar_getsample` function, pass it to the `Exemple` function, get back the output which is then transferred to the polynomial `pc` by the `polychaos_settarget` function.

```
// This is slow.
for k=1:np
  inputdata = setrandvar_getsample(srvu,k);
  outputdata = Exemple(inputdata);
  mprintf ( "Experiment␣#%d,␣input␣=[%f␣%f],␣output␣=␣%f\n", k, ..
    inputdata(1), inputdata(2) , outputdata )
  polychaos_settarget(pc,k,outputdata);
end
```

The previous script produces the following output.

```
Experiment #1, input =[0.133975 1.169052], output = 0.156623
Experiment #2, input =[0.133975 1.750000], output = 0.234456
Experiment #3, input =[0.133975 2.330948], output = 0.312288
Experiment #4, input =[1.000000 1.169052], output = 1.169052
Experiment #5, input =[1.000000 1.750000], output = 1.750000
Experiment #6, input =[1.000000 2.330948], output = 2.330948
Experiment #7, input =[1.866025 1.169052], output = 2.181482
Experiment #8, input =[1.866025 1.750000], output = 3.265544
Experiment #9, input =[1.866025 2.330948], output = 4.349607
```

There is actually a much faster way of computing the output. Indeed, using vectorisation, we can compute all the outputs in one single call to the `Exemple` function.

```
// This is fast.
inputdata = setrandvar_getsample(srvu);
outputdata = Exemple(inputdata);
polychaos_settarget(pc,outputdata);
```

We can compute the polynomial expansion based on numerical integration so that the coefficients of the polynomial are determined. This is done with the `polychaos_computeexp` function, which stands for "compute the expansion".

```
polychaos_setdegree(pc,degre);
polychaos_computeexp(pc,srvx,"Integration");
```

Everything is now ready for the sensitivity analysis. Indeed, the `polychaos_getmean` returns the mean while the `polychaos_getvariance` returns the variance.

```
average = polychaos_getmean(pc);
var = polychaos_getvariance(pc);
mprintf("Mean␣␣␣␣=␣%f\n",average);
mprintf("Variance␣␣␣␣=␣%f\n",var);
mprintf("Indice␣de␣sensibilite␣du␣1er␣ordre\n");
mprintf("␣␣␣␣Variable␣X1␣=␣%f\n",polychaos_getindexfirst(pc,1));
mprintf("␣␣␣␣Variable␣X2␣=␣%f\n",polychaos_getindexfirst(pc,2));
mprintf("Indice␣de␣sensibilite␣Totale\n");
```

```
mprintf("    Variable X1 = %f\n",polychaos_getindextotal(pc,1));
mprintf("    Variable X2 = %f\n",polychaos_getindextotal(pc,2));
```

The previous script produces the following output.

```
Mean     = 1.750000
Variance     = 1.000000
Indice de sensibilite du 1er ordre
    Variable X1 = 0.765625
    Variable X2 = 0.187500
Indice de sensibilite Totale
    Variable X1 = 0.812500
    Variable X2 = 0.234375
```

In order to free the memory required for the computation, it is necessary to delete all the objects created so far.

```
polychaos_destroy(pc);
randvar_destroy(vu1);
randvar_destroy(vu2);
randvar_destroy(vx1);
randvar_destroy(vx2);
setrandvar_destroy(srvu);
setrandvar_destroy(srvx);
```

Prior to destroying the objects, we can inquire a little more about the density of the output of the chaos polynomial. In the following script, we create a Latin Hypercube Sampling made of 10 000 points. Then get the output of the polynomial on these inputs and plot the histogram of the output.

```
polychaos_buildsample(pc,"Lhs",10000,0);
sample_output = polychaos_getsample(pc);
scf();
histplot(50,sample_output);
xtitle("Product function - Empirical Histogram","X","P(X)");
```

The previous script produces the figure 6.3.

We may explore the following topics.

- Perform the same analysis where the variable X_2 is a normal variable with mean 2 and standard deviation 2.
- Check that the development in polynomial chaos on a Hermite-Hermite basis does not allow to get exact results. See that the convergence can be obtained by increasing the degree.
- Check that the development on a basis Hermite-Legendre allows to get exact results with degree 2.

6.2.2 The Ishigami test case

In this section, we present the Ishigami test case.

The function `Exemple` is the model that we consider in this numerical experiment. This function takes a vector of size 3 in input and returns a scalar output.

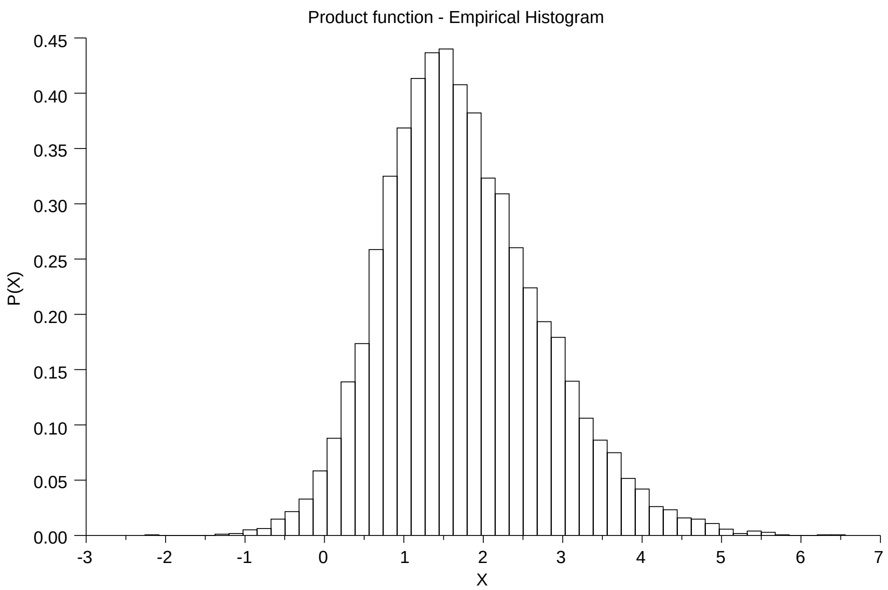

Figure 6.3: Product function - Histogram of the output on a LHS design with 10000 experiments.

```
function y = Exemple (x)
  a=7.
  b=0.1
  s1=sin(x(:,1))
  s2=sin(x(:,2))
  y(:,1) = s1 + a.*s2.*s2 + b.*x(:,3).*x(:,3).*x(:,3).*x(:,3).*s1
endfunction
```

We create 3 uncertain parameters which are uniform in the interval $[-\pi, \pi]$ and put these random variables into the collection `srvu`.

```
rvu1 = randvar_new("Uniforme",-%pi,%pi);
rvu2 = randvar_new("Uniforme",-%pi,%pi);
rvu3 = randvar_new("Uniforme",-%pi,%pi);

srvu = setrandvar_new();
setrandvar_addrandvar ( srvu, rvu1);
setrandvar_addrandvar ( srvu, rvu2);
setrandvar_addrandvar ( srvu, rvu3);
```

The collection of stochastic variables is created with the function `setrandvar_new`. The calling sequence `srvx = setrandvar_new( nx )` allows to create a collection of `nx=3` random variables uniform in the interval $[0, 1]$. Then we create a Petras DOE for the stochastic collection `srvx` and transform it into a DOE for the uncertain parameters `srvu`.

```
nx = setrandvar_getdimension ( srvu );
srvx = setrandvar_new( nx );
degre = 9;
setrandvar_buildsample(srvx,"Petras",degre);
```

```
setrandvar_buildsample( srvu , srvx );
```

We use the polychaos_new function and create the new polynomial pc with 3 inputs and 1 output.

```
noutput = 1;
pc = polychaos_new ( srvx , noutput );
```

The next step allows to perform the simulations associated with the DOE prescribed by the collection srvu. Here, we must perform np=751 experiments.

```
np = setrandvar_getsize(srvu);
polychaos_setsizetarget(pc,np);
// This is slow
for k=1:np
  inputdata = setrandvar_getsample(srvu,k);
  outputdata = Exemple(inputdata);
  polychaos_settarget(pc,k,outputdata);
end
```

The previous loop works, but is slow when np is large. Instead, the following script is fast because it uses vectorization.

```
// This is fast
inputdata = setrandvar_getsample(srvu);
outputdata = Exemple(inputdata);
polychaos_settarget(pc,outputdata);
```

We can now compute the polynomial expansion by integration.

```
polychaos_setdegree(pc,degre);
polychaos_computeexp(pc,srvx,"Integration");
```

Everything is now ready so that we can do the sensitivy analysis, as in the following script.

```
average = polychaos_getmean(pc);
var = polychaos_getvariance(pc);
mprintf("Mean␣␣␣␣␣␣␣␣=␣%f\n",average);
mprintf("Variance␣␣␣␣=␣%f\n",var);
mprintf("First␣order␣sensitivity␣index\n");
mprintf("␣␣␣␣Variable␣X1␣=␣%f\n",polychaos_getindexfirst(pc,1));
mprintf("␣␣␣␣Variable␣X2␣=␣%f\n",polychaos_getindexfirst(pc,2));
mprintf("␣␣␣␣Variable␣X3␣=␣%f\n",polychaos_getindexfirst(pc,3));
mprintf("Total␣sensitivity␣index\n");
mprintf("␣␣␣␣Variable␣X1␣=␣%f\n",polychaos_getindextotal(pc,1));
mprintf("␣␣␣␣Variable␣X2␣=␣%f\n",polychaos_getindextotal(pc,2));
mprintf("␣␣␣␣Variable␣X3␣=␣%f\n",polychaos_getindextotal(pc,3));
```

The previous script produces the following output.

```
Mean        = 3.500000
Variance    = 13.842473
First order sensitivity index
    Variable X1 = 0.313953
    Variable X2 = 0.442325
    Variable X3 = 0.000000
```

```
Total sensitivity index
    Variable X1 = 0.557675
    Variable X2 = 0.442326
    Variable X3 = 0.243721
```

We now focus on the variance generated by the variables #1 and #3. We set the group to the empty group with the `polychaos_setgroupempty` function and add variables with the `polychaos_setgroupaddvar` function.

```
groupe = [1 3];
polychaos_setgroupempty ( pc );
polychaos_setgroupaddvar ( pc , groupe(1) );
polychaos_setgroupaddvar ( pc , groupe(2) );
mprintf("Fraction of the variance of a group of variables\n");
mprintf("    Groupe X1 et X2 =%f\n",polychaos_getgroupind(pc));
```

The previous script produces the following output.

```
Fraction of the variance of a group of variables
    Groupe X1 et X2 =0.557674
```

The function `polychaos_getanova` prints the functionnal decomposition of the normalized variance.

```
polychaos_getanova(pc);
```

The previous script produces the following output.

```
1 0 0  : 0.313953
0 1 0  : 0.442325
1 1 0  : 1.55229e-009
0 0 1  : 8.08643e-031
1 0 1  : 0.243721
0 1 1  : 7.26213e-031
1 1 1  : 1.6007e-007
```

We can compute the density function associated with the output variable of the function. In order to compute it, we use the `polychaos_buildsample` function and create a Latin Hypercube Sampling with 10000 experiments. The `polychaos_getsample` function allows to quiery the polynomial and get the outputs. We plot it with the `histplot` Scilab graphic function, which produces the figure 6.4.

```
polychaos_buildsample(pc,"Lhs",10000,0);
sample_output = polychaos_getsample(pc);
scf();
histplot(50,sample_output)
xtitle("Ishigami - Histogram");
```

We can plot a bar graph of the sensitivity indices, as presented in figure 6.5.

```
for i=1:nx
  indexfirst(i)=polychaos_getindexfirst(pc,i);
  indextotal(i)=polychaos_getindextotal(pc,i);
end
scf();
```

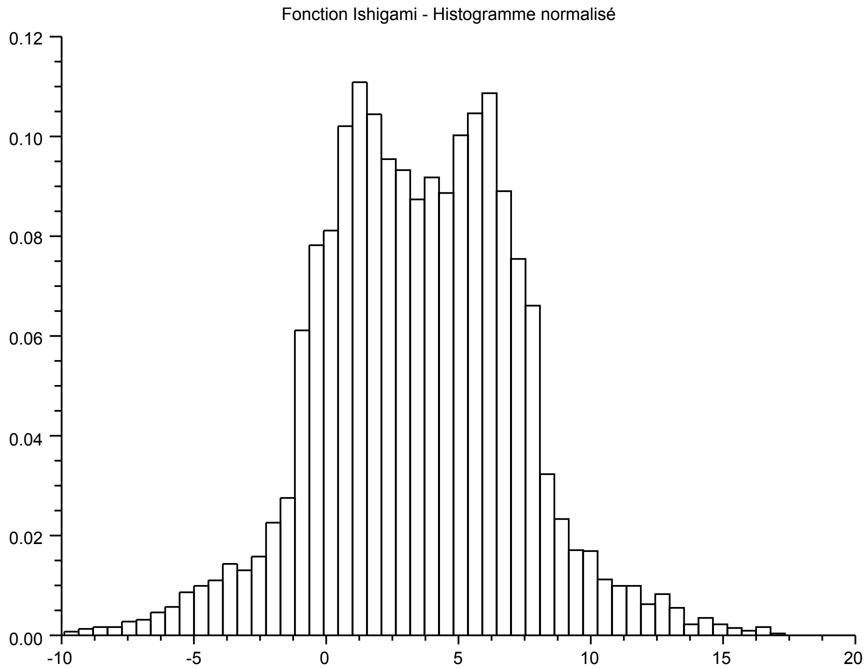

Figure 6.4: Ishigami function - Histogram of the variable on a LHS design with 10000 experiments.

```
bar(indextotal,0.2,'blue');
bar(indexfirst,0.15,'yellow');
legend(["Total" "First order"],pos=1);
xtitle("Ishigami - Sensitivity indices");
```

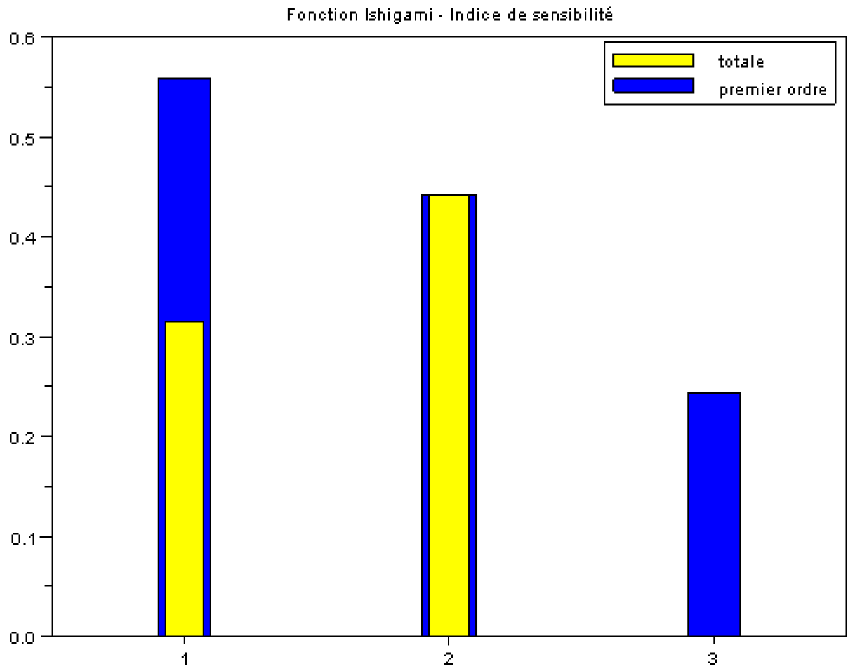

Figure 6.5: Ishigami function - Sensitivity indices.

Chapter 7

A tutorial introduction to sensitivity analysis

In this chapter, we (extremely) briefly present the theory which is used in the library. This section is a tutorial introduction to the NISP module.

7.1 Sensitivity analysis

In this section, we present the sensitivity analysis and emphasize the difference between global and local analysis.

Consider the model

$$\mathbf{Y} = f(\mathbf{X}), \tag{7.1}$$

where $\mathbf{X} \in D_X \subset \mathbb{R}^p$ is the input and $\mathbf{Y} \in D_Y \subset \mathbb{R}^m$ is the output of the model. The mapping f is presented in figure 7.1.

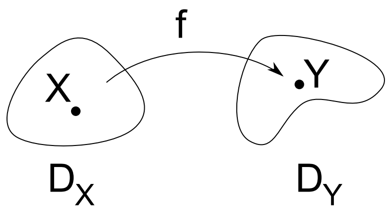

Figure 7.1: Global analysis.

The assume that the input $\mathbf{X}$ is a random variable, so that the output variable $\mathbf{Y}$ is also a random variable. We are interested in measuring the sensitivity of the output depending on the uncertainty of the input. More precisely, we are interested in knowing

- the input variables X_i which generate the most variability in the output $\mathbf{Y}$,
- the input variables X_i which are not significant,
- a sub-space of the input variables where the variability is maximum,

- if input variables interacts.

Consider the mapping presented in figure 7.1. The f mapping transforms the domain D_X into the domain D_Y. If f is sufficiently smooth, small perturbations of X generate small perturbations of Y. The local sensitivity analysis focuses on the behaviour of the mapping in the neighbourhood of a particular point $X \in D_X$ toward a particular point $Y \in D_Y$. The global sensisitivity analysis models the propagation of uncertainties transforming the whole set D_X into the set D_Y.

In the following, we assume that there is only one output variable so that $Y \in \mathbb{R}$.

There are two types of analysis that we are going to perform, that is uncertainty analysis and sensitivity analysis.

In uncertainty analysis, we assume that f_X is the probability density function of the variable $\mathbf{X}$ and we are searching for the probability density function f_Y of the variable Y and by its cumulated density function $F_Y(y) = P(Y \leq y)$. This problem is difficult in the general case, and this is why we often are looking for an estimate of the expectation of Y, as defined by

$$E(Y) = \int_{D_X} y f_Y(y) dy, \tag{7.2}$$

and an estimate of its variance

$$Var(Y) = \int_{D_X} (y - E(Y)) f_Y(y) dy. \tag{7.3}$$

We might also be interested in the computation of the probability over a threshold.

In sensitivity analysis, we focus on the relative importance of the input variable X_i on the uncertainty of $\mathbf{Y}$. This way, we can order the input variables so that we can reduce the variability of the most important input variables, in order to, finally, reduce the variability of $\mathbf{Y}$.

More details on this topic can be found in the papers of Homma and Saltelli [1] or in the work of Sobol [4]. The Thesis by Jacques [2] presents an overview of sensitivity analysis.

7.2 Standardized regression coefficients of affine models

In this section, we present the standardized regression coefficients of an affine model.

Assume that the random variables X_i are independent, with mean $E(X_i)$ and finite variances $V(X_i)$, for $i = 1, 2, \ldots, p$. Let us consider the random variable Y as an affine function of the variables X_i:

$$Y = \beta_0 + \sum_{i=1,2,\ldots,p} \beta_i X_i, \tag{7.4}$$

where β_i are real parameters, for $i = 1, 2, \ldots, p$.

The expectation of the random variable Y is

$$E(Y) = \beta_0 + \sum_{i=1,2,\ldots,p} \beta_i E(X_i). \tag{7.5}$$

Since the variables X_i are independent, the variance of the sum of variables is the sum of the variances. Hence,

$$V(Y) = V(\beta_0) + \sum_{i=1,2,\ldots,p} V(\beta_i X_i), \tag{7.6}$$

which leads to the equality

$$V(Y) = \sum_{i=1,2,\ldots,p} \beta_i^2 V(X_i). \quad (7.7)$$

Hence, each term $\beta_i^2 V(X_i)$ is the part of the total variance $V(Y)$ which is caused by the variable X_i.

The standardized regression coefficient is defined as

$$SRC_i = \frac{\beta_i^2 V(X_i)}{V(Y)}, \quad (7.8)$$

for $i = 1, 2, \ldots, p$.

Hence, the sum of the standardized regression coefficients is one:

$$SRC_1 + SRC_2 + \ldots + SRC_p = 1. \quad (7.9)$$

7.3 Link with the linear correlation coefficients

In this section, we present the link between the linear correlation coefficients of an affine model, and the standardized regression coefficients.

Assume that the random variables X_i are independent, with mean $E(X_i)$ and finite variances $V(X_i)$, for $i = 1, 2, \ldots, p$. Let us consider the random variable Y, which depends linearily on the variables X_i by the relationship 7.4.

The linear correlation coefficient between Y and X_i is

$$\rho_{Y,X_i} = \frac{Cov(Y, X_i)}{\sqrt{V(Y)}\sqrt{V(X_i)}}, \quad (7.10)$$

for $i = 1, 2, \ldots, p$. In the particular case of the affine model 7.4, we have

$$\begin{aligned} Cov(Y, X_i) &= Cov(\beta_0, X_i) + \beta_1 Cov(X_1, X_i) + \beta_2 Cov(X_2, X_i) + \ldots + && (7.11) \\ & \quad \beta_i Cov(X_i, X_i) + \ldots + \beta_p Cov(X_p, X_i). && (7.12) \\ & && (7.13) \end{aligned}$$

Since the random variables X_i are independent, we have $Cov(X_j, X_i) = 0$, for any $j \neq i$. Therefore,

$$\begin{aligned} Cov(Y, X_i) &= \beta_i Cov(X_i, X_i) && (7.14) \\ &= \beta_i V(X_i). && (7.15) \end{aligned}$$

Hence, the correlation coefficient can be simplified into

$$\begin{aligned} \rho_{Y,X_i} &= \frac{\beta_i V(X_i)}{\sqrt{V(Y)}\sqrt{V(X_i)}} && (7.16) \\ &= \frac{\beta_i \sqrt{V(X_i)}}{\sqrt{V(Y)}}. && (7.17) \end{aligned}$$

We square the previous equality and get

$$\rho^2_{Y,X_i} = \frac{\beta_i^2 V(X_i)}{V(Y)}. \tag{7.18}$$

Therefore, the square of the linear correlation coefficient is equal to the first order sensitivity index, i.e.

$$\rho^2_{Y,X_i} = SRC_i. \tag{7.19}$$

7.4 Using scatter plots

In this section, we present an example of an affine model, where the difference between local and global sensitivity is made clearer by the use of scatter plots.

Assume four independent random variables X_i, for $i = 1, 2, 3, 4$. We assume that the variables X_i are normally distributed, with zero mean and i^2 variance.

Let us consider the affine model

$$Y = X_1 + X_2 + X_3 + X_4. \tag{7.20}$$

Notice that the derivative of Y with respect to any of its input is equal to one, i.e.

$$\frac{\partial Y}{\partial X_i} = 1, \tag{7.21}$$

for $i = 1, 2, 3, 4$. This means that, locally, the inputs all have the same effect on the output.

Let us compute the standardized regression coefficients of this model. By hypothesis, the variance of each variable is

$$V(X_1) = 1, \qquad V(X_2) = 4, \tag{7.22}$$

$$V(X_3) = 9, \qquad V(X_4) = 16. \tag{7.23}$$

Since the variables are independent, the variance of the output Y is

$$V(Y) = V(X_1) + V(X_2) + V(X_3) + V(X_4) = 30. \tag{7.24}$$

The standardized regression coefficient is

$$SRC_i = \frac{\beta_i^2 V(X_i)}{V(Y)} = \frac{i^2}{30}, \tag{7.25}$$

for $i = 1, 2, 3, 4$. More specifically, we have

$$SRC_1 = \frac{1}{30}, \qquad SRC_2 = \frac{4}{30}, \tag{7.26}$$

$$SRC_3 = \frac{9}{30}, \qquad SRC_4 = \frac{16}{30}. \tag{7.27}$$

We have the following inequalities:

$$SRC_4 > SRC_3 > SRC_2 > SRC_1. \tag{7.28}$$

This means that the variable which causes the most variance in the output is X_4, while the variable which causes the least variance in the output is X_1.

The script below performs the analysis with the NISP module. The sampling is based on a Latin Hypercube Sampling design with 5000 points.

```
function y = Exemple (x)
  y(:) = x(:,1) + x(:,2) + x(:,3) + x(:,4)
endfunction

function r = lincorrcoef ( x , y )
    // Returns the linear correlation coefficient of x and y.
    // The variables are expected to be column matrices with the same size.
    x = x(:)
    y = y(:)
    mx = mean(x)
    my = mean(y)
    sx = sqrt(sum((x-mx).^2))
    sy = sqrt(sum((y-my).^2))
    r = (x-mx)'*(y-my) / sx / sy
endfunction

// Initialisation de la graine aleatoire
nisp_initseed ( 0 );

// Create the random variables.
rvu1 = randvar_new("Normale",0,1);
rvu2 = randvar_new("Normale",0,2);
rvu3 = randvar_new("Normale",0,3);
rvu4 = randvar_new("Normale",0,4);
srvu = setrandvar_new();
setrandvar_addrandvar ( srvu, rvu1);
setrandvar_addrandvar ( srvu, rvu2);
setrandvar_addrandvar ( srvu, rvu3);
setrandvar_addrandvar ( srvu, rvu4);
// Create a sampling by a Latin Hypercube Sampling with size 5000.
nbshots = 5000;
setrandvar_buildsample(srvu, "Lhs",nbshots);
sampling = setrandvar_getsample(srvu);
// Perform the experiments.
y=Exemple(sampling);
// Scatter plots : y depending on X_i
for k=1:4
  scf();
  plot(y,sampling(:,k),'rx');
  xistr="X"+string(k);
  xtitle("Scatter␣plot␣for␣"+xistr,xistr,"Y");
end
// Compute the sample linear correlation coefficients
rho1 = lincorrcoef ( sampling(:,1) , y );
SRC1 = rho1^2;
SRC1expected = 1/30;
mprintf("SRC_1=%.5f␣(expected=%.5f)\n",SRC1,SRC1expected);
//
rho2 = lincorrcoef ( sampling(:,2) , y );
```

```
SRC2 = rho2^2;
SRC2expected = 4/30;
mprintf("SRC_2=%.5f␣(expected=%.5f)\n",SRC2,SRC2expected);
//
rho3 = lincorrcoef ( sampling(:,3) , y );
SRC3 = rho3^2;
SRC3expected = 9/30;
mprintf("SRC_3=%.5f␣(expected=%.5f)\n",SRC3,SRC3expected);
//
rho4 = lincorrcoef ( sampling(:,4) , y );
SRC4 = rho4^2;
SRC4expected = 16/30;
mprintf("SRC_4=%.5f␣(expected=%.5f)\n",SRC4,SRC4expected);
//
SUM = SRC1 + SRC2 + SRC3 + SRC4;
SUMexpected = 1;
mprintf("SUM=%.5f␣(expected=%.5f)\n",SUM,SUMexpected);
//
// Clean-up
randvar_destroy(rvu1);
randvar_destroy(rvu2);
randvar_destroy(rvu3);
randvar_destroy(rvu4);
setrandvar_destroy(srvu);
```

The previous script produces the following output.

```
SRC_1=0.03538 (expected=0.03333)
SRC_2=0.12570 (expected=0.13333)
SRC_3=0.31817 (expected=0.30000)
SRC_4=0.54314 (expected=0.53333)
SUM=1.00066 (expected=1.00000)
```

The previous script also produces the plots 7.2, 7.3, 7.4 and 7.5.

7.5 Sensitivity analysis for nonlinear models

Let us focus on one particular input X_i of the model f, with $i = 1, 2, \ldots, p$. If we set X_i to a particular value, say x_i for example, then the variance of the output Y surely decreases, because the variable X_i is not randome anymore. We can then measure the conditionnal variance given X_i, denoted by $V(Y|X_i = x_i)$.

Since X_i is a random variable, the conditionnal variance $V(Y|X_i = x_i)$ is a random variable. We are interested in the average value of this variance, that is, we are interested in $E(V(Y|X_i))$. If X_i has a large weight in the variance $V(Y)$, then $E(V(Y|X_i))$ is small.

The theorem of the total variance states that, if $V(Y)$ is finite, then

$$V(Y) = V(E(Y|X_i)) + E(V(Y|X_i)). \tag{7.29}$$

If X_i has a large weight in the variance $V(Y)$, then $V(E(Y|X_i))$ is large.

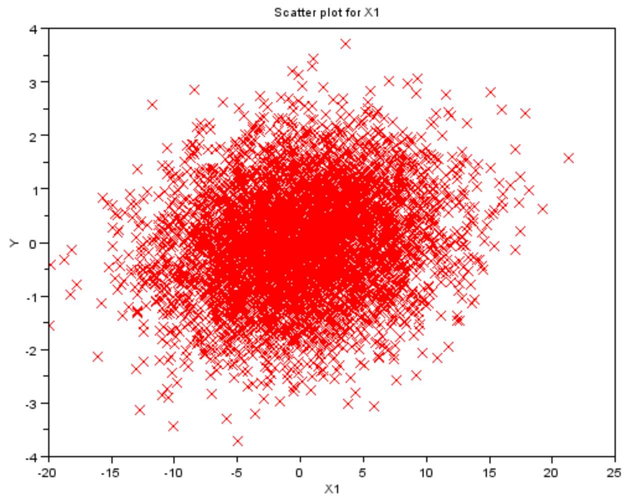

Figure 7.2: Scatter plot for an affine model - Variable X_1.

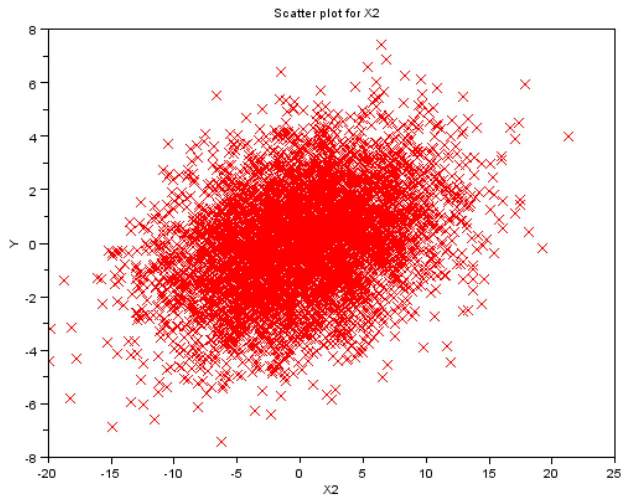

Figure 7.3: Scatter plot for an affine model - Variable X_2.

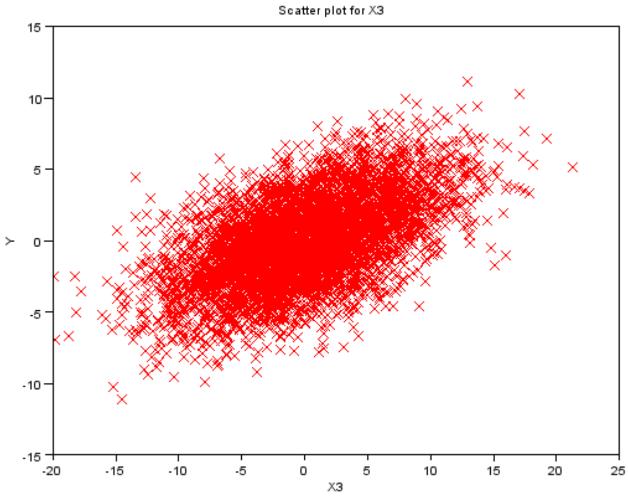

Figure 7.4: Scatter plot for an affine model - Variable X_3.

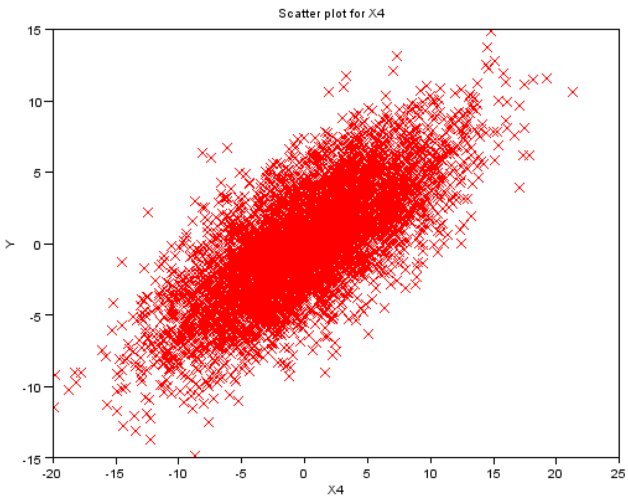

Figure 7.5: Scatter plot for an affine model - Variable X_4.

The first order sensivity indice of Y to the variable X_i is defined by

$$S_i = \frac{V(E(Y|X_i))}{V(Y)}, \tag{7.30}$$

for $i = 1, 2, \ldots, p$. The sensitivity indice measures the part of the variance which is caused by the uncertainty in X_i.

We can compute the sensitivity indice when the function f is linear. Assume that the output Y depends linearily on the input X_i:

$$Y = \beta_0 + \sum_{i=1,2,\ldots,p} \beta_i X_i, \tag{7.31}$$

where $\beta_i \in \mathbb{R}$, for $i = 0, 1, 2, \ldots, p$. Then

$$E(Y|X_i) = \beta_0 + \sum_{i=1,2,\ldots,i-1,i+1,\ldots,p} \beta_i E(X_i) + \beta_i X_i, \tag{7.32}$$

since the expection of a sum is the sum of expectations. Then,

$$\begin{aligned} V(E(Y|X_i)) &= V(\beta_i X_i) && (7.33) \\ &= \beta_i^2 V(X_i), && (7.34) \end{aligned}$$

since the variance of a constant term is zero. Therefore, the sensitivity index of Y to the variable X_i is

$$S_i = \frac{\beta_i^2 V(X_i)}{V(Y)}, \tag{7.35}$$

for $i = 1, 2, \ldots, p$.

Hence, if we make the assumption that a model is affine, then the empirical linear correlation coefficient can be used to estimate the sensitivity indices.

7.6 The effect of the interactions

In this example, we consider a non-linear, non-additive model made of the product of two independent random variables. The goal of this example is to show that, in some cases, we have to consider the interations between the variables.

Consider the function

$$Y = X_1 X_2, \tag{7.36}$$

where X_1 and X_2 are two independent normal random variables with mean μ_1 and μ_2 and variances σ_1^2 and σ_2^2.

Let us compute the expectation of the random variable Y. The expectation of Y is

$$E(Y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} X_1 X_2 F(X_1, X_2) dx_1 dx_2, \tag{7.37}$$

where $F(x_1, x_2)$ is the joint probability distribution function of the variables X_1 and X_2. Since X_1 and X_2 are independent variables, we have

$$F(x_1, x_2) = F_1(X_1)F_2(X_2), \tag{7.38}$$

where F_1 is the probability distribution function of X_1 and F_2 is the probability distribution function of X_2. Then, we have

$$\begin{aligned} E(Y) &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} X_1X_2F_1(X_1)F_2(X_2)dx_1dx_2 && (7.39)\\ &= \left(\int_{-\infty}^{\infty} X_1F_1(X_1)dx_1\right)\left(\int_{-\infty}^{\infty} X_2F_2(X_2)dx_2\right). && (7.40)\\ &= E(X_1)E(X_2). && (7.41) \end{aligned}$$

Therefore,

$$E(Y) = \mu_1\mu_2. \tag{7.42}$$

The variance of Y is

$$V(Y) = E(Y^2) - E(Y)^2. \tag{7.43}$$

The expectation $E(Y^2)$ is

$$\begin{aligned} E(Y^2) &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} (X_1X_2)^2F(x_1, x_2)dx_1dx_2 && (7.44)\\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} (X_1X_2)^2F_1(X_1)F_2(X_2)dx_1dx_2 && (7.45)\\ &= \left(\int_{-\infty}^{\infty} X_1^2F_1(X_1)dx_1\right)\left(\int_{-\infty}^{\infty} X_2^2F_2(X_2)dx_2\right), && (7.46)\\ &= E(X_1^2)E(X_2^2). && (7.47) \end{aligned}$$

Now, we have

$$V(X_1) = E(X_1^2) - E(X_1)^2, \tag{7.48}$$

$$V(X_2) = E(X_2^2) - E(X_2)^2, \tag{7.49}$$

which leads to

$$E(X_1^2) = V(X_1) + E(X_1)^2, \tag{7.50}$$

$$E(X_2^2) = V(X_2) + E(X_2)^2. \tag{7.51}$$

Therefore,

$$\begin{aligned} E(Y^2) &= (V(X_1) + E(X_1)^2)(V(X_2) + E(X_2)^2) && (7.52)\\ &= (\sigma_1^2 + \mu_1^2)(\sigma_2^2 + \mu_2^2). && (7.53) \end{aligned}$$

Finally, we get

$$V(Y) = (\sigma_1^2 + \mu_1^2)(\sigma_2^2 + \mu_2^2) - (\mu_1\mu_2)^2. \tag{7.54}$$

We can expand the previous equality and get

$$V(Y) = \sigma_1^2\sigma_2^2 + \sigma_1^2\mu_2^2 + \mu_1^2\sigma_2^2 + \mu_1^2\mu_2^2 - (\mu_1\mu_2)^2. \tag{7.55}$$

The last two terms of the previous equality can be simplified, so that we get

$$V(Y) = \sigma_1^2\sigma_2^2 + \sigma_1^2\mu_2^2 + \mu_1^2\sigma_2^2. \tag{7.56}$$

The sensitivity indices can be computed from the definitions

$$S_1 = \frac{V(E(Y|X_1))}{V(Y)}, \tag{7.57}$$

$$S_2 = \frac{V(E(Y|X_2))}{V(Y)}. \tag{7.58}$$

We have $E(Y|X_1) = E(X_2)X_1 = \mu_2 X_1$. Similarily, $E(Y|X_2) = \mu_1 X_2$. Hence

$$S_1 = \frac{V(\mu_2 X_1)}{V(Y)}, \tag{7.59}$$

$$S_2 = \frac{V(X_2)}{V(Y)}. \tag{7.60}$$

We get

$$S_1 = \frac{\mu_2^2 V(X_1)}{V(Y)}, \tag{7.61}$$

$$S_2 = \frac{\mu_1^2 V(X_2)}{V(Y)}. \tag{7.62}$$

Finally, the first order sensitivity indices are

$$S_1 = \frac{\mu_2^2\sigma_1^2}{V(Y)}, \tag{7.63}$$

$$S_2 = \frac{\mu_1^2\sigma_2^2}{V(Y)}. \tag{7.64}$$

Since $\sigma_1^2\sigma_2^2 \geq 0$, we have

$$\mu_2^2\sigma_1^2 + \mu_1^2\sigma_2^2 \leq V(Y) = \sigma_1^2\sigma_2^2 + \sigma_1^2\mu_2^2 + \mu_1^2\sigma_2^2. \tag{7.65}$$

We divide the previous inequality by $V(Y)$, and get

$$\frac{\mu_2^2\sigma_1^2}{V(Y)} + \frac{\mu_1^2\sigma_2^2}{V(Y)} \leq 1. \tag{7.66}$$

Therefore, the sum of the first order sensitivity indices satisfies the inequality

$$S_1 + S_2 \leq 1. \tag{7.67}$$

Hence, in this example, one part of the variance $V(Y)$ cannot be explained neither by X_1 alone, or by X_2 alone, because it is caused by the interactions between X_1 and X_2. We define by S_{12} the sensitivity index associated with the group of variables (X_1, X_2) as

$$S_{12} = 1 - S_1 - S_2 = \frac{\sigma_1^2 \sigma_2^2}{V(Y)}. \tag{7.68}$$

The following Scilab script performs the sensitivity analysis on the previous example. We consider two normal variables, where the first variable has mean 1.5 and standard deviation 0.5 while the second variable has mean 3.5 and standard deviation 2.5.

```
function y = Exemple (x)
  y(:,1) = x(:,1) .* x(:,2);
endfunction
// First variable
// Normal
mu1 = 1.5;
sigma1 = 0.5;
// Second variable
// Normal
mu2 = 3.5;
sigma2 = 2.5;

// 1. Two stochastic (normalized) variables
vx1 = randvar_new("Normale");
vx2 = randvar_new("Normale");
// 2. A collection of stoch. variables.
srvx = setrandvar_new();
setrandvar_addrandvar ( srvx,vx1);
setrandvar_addrandvar ( srvx,vx2);
// 3. Two uncertain parameters
vu1 = randvar_new("Normale",mu1,sigma1);
vu2 = randvar_new("Normale",mu2,sigma2);
// 4. A collection of uncertain parameters
srvu = setrandvar_new();
setrandvar_addrandvar ( srvu,vu1);
setrandvar_addrandvar ( srvu,vu2);
// 5. Create the Design Of Experiments
degre = 2;
setrandvar_buildsample(srvx,"Quadrature",degre);
setrandvar_buildsample( srvu , srvx );
// 6. Create the polynomial
ny = 1;
pc = polychaos_new ( srvx , ny );
np = setrandvar_getsize(srvx);
mprintf("Number␣of␣experiments␣:␣%d\n",np);
polychaos_setsizetarget(pc,np);
// 7. Perform the DOE
inputdata = setrandvar_getsample(srvu);
outputdata = Exemple(inputdata);
```

```
polychaos_settarget(pc,outputdata);
// 8. Compute the coefficients of the polynomial expansion
polychaos_setdegree(pc,degre);
polychaos_computeexp(pc,srvx,"Integration");
// 9. Get the sensitivity indices
average = polychaos_getmean(pc);
Ey_expectation= mu1*mu2;
var = polychaos_getvariance(pc);
Vy_expectation = mu2^2*sigma1^2 + mu1^2*sigma2^2 + sigma1^2*sigma2^2;
mprintf("Mean␣␣␣␣=␣%f␣(expectation␣=␣%f)\n",average,Ey_expectation);
mprintf("Variance␣␣␣␣=␣%f␣(expectation␣=␣%f)\n",var,Vy_expectation);
mprintf("First␣order␣sensitivity␣index\n");
S1 = polychaos_getindexfirst(pc,1);
S1_expectation = ( mu2^2*sigma1^2 ) / Vy_expectation;
mprintf("␣␣␣␣Variable␣X1␣=␣%f␣(expectation␣=␣%f)\n",S1,S1_expectation);
re = abs(S1- S1_expectation)/S1_expectation;
mprintf("␣␣␣␣␣␣␣␣Relative␣Error␣=␣%f\n", re);
S2 = polychaos_getindexfirst(pc,2);
S2_expectation = ( mu1^2*sigma2^2 ) / Vy_expectation;
mprintf("␣␣␣␣Variable␣X2␣=␣%f␣(expectation␣=␣%f)\n",S2,S2_expectation);
re = abs(S2- S2_expectation)/S2_expectation;
mprintf("␣␣␣␣␣␣␣␣Relative␣Error␣=␣%f\n", re);

mprintf("Total␣sensitivity␣index\n");
ST1 = polychaos_getindextotal(pc,1);
mprintf("␣␣␣␣Variable␣X1␣=␣%f\n",ST1);
ST2 = polychaos_getindextotal(pc,2);
mprintf("␣␣␣␣Variable␣X2␣=␣%f\n",ST2);
// Clean-up
polychaos_destroy(pc);
randvar_destroy(vu1);
randvar_destroy(vu2);
randvar_destroy(vx1);
randvar_destroy(vx2);
setrandvar_destroy(srvu);
setrandvar_destroy(srvx);
```

The previous script produces the following output.

```
Mean    = 5.250000 (expectation = 5.250000)
Variance    = 18.687500 (expectation = 18.687500)
First order sensitivity index
    Variable X1 = 0.163880 (expectation = 0.163880)
        Relative Error = 0.000000
    Variable X2 = 0.752508 (expectation = 0.752508)
        Relative Error = 0.000000
Total sensitivity index
    Variable X1 = 0.247492
    Variable X2 = 0.836120
```

We see that the polynomial chaos performs an exact computation.

7.7 Sobol decomposition

Sobol [4] introduced the sensitivity index based on $V(E(Y|X_i))$ by decomposing the function f as a sum of function with an increasing number of parameters.

We consider the function f

$$Y = f(x_1, x_2, \ldots, x_p), \tag{7.69}$$

where $x = (x_1, \ldots, x_p) \in [0,1]^p$. If f can be integrated in $[0,1]^p$, then there is a unique decomposition

$$Y = f_0 + \sum_{i=1,2,\ldots,p} f_i(x_i) + \sum_{1 \leq i < j \leq p} f_{i,j}(x_i, x_j) + \ldots + f_{1,2,\ldots,p}(x_1, x_2, \ldots, x_p), \tag{7.70}$$

where f_0 is a constant and the function of the decomposition satisfy the equalities

$$\int_0^1 f_{i_1,\ldots,i_s}(x_{i_1}, \ldots, x_{i_s}) dx_{i_k} = 0, \tag{7.71}$$

for any $k = 1, 2, \ldots, s$ and any indices $\{i_1, \ldots, i_s\} \subset \{1, \ldots, p\}$.

The equalities 7.71 mean that the integral of each function with respect to one of its variables is zero. The immediate consequence of this is that the decomposition functions are orthogonal, i.e.

$$\int_0^1 f_{i_1,\ldots,i_s}(x_{i_1}, \ldots, x_{i_s}) f_{j_1,\ldots,j_s}(x_{j_1}, \ldots, x_{j_s}) dx_1 \ldots dx_p = 0, \tag{7.72}$$

if $(i_1, \ldots, i_s) \neq (j_1, \ldots, j_s)$.

This because if the two set of indices $(i_1, \ldots, i_s)$ and $(j_1, \ldots, j_s)$, this means that there is at least one index k which appears in one index and not in the other. By the equality 7.71, this implies that if we integrate with respect to x_k, then the integral is zero. Since the integral in 7.72 is for all the variables, since implies that all the integral is zero.

We are now going to explicitely compute the decomposition functions f_0, f_i, $f_{i,j}$, etc... by integration the decomposition, using the orthogonality to simplify the results. If we integrate the equation 7.70 with respect to all the variables, we get

$$\int_0^1 f(x) dx = f_0. \tag{7.73}$$

If we integrate the equation 7.70 with respect to all the variables except i, we get

$$\int_0^1 f(x) dx_{\sim i} = f_0 + f_i(x_i), \tag{7.74}$$

where $x_{\sim i}$ is the vector x without its i-th component, i.e.

$$x_{\sim i} = (x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_p). \tag{7.75}$$

If we integrate the equation 7.70 with respect to all the variables except i and j, we get

$$\int_0^1 f(x) dx_{\sim i,j} = f_0 + f_i(x_i) + f_j(x_j) + f_{i,j}(x_i, x_j). \tag{7.76}$$

If we integrate the equation 7.70 with respect to all the variables except i and j and k, we get

$$\begin{aligned}\int_0^1 f(x)dx_{\sim i,j,k} &= f_0 + f_i(x_i) + f_j(x_j) + f_k(x_k) + \quad (7.77)\\ & \quad f_{i,j}(x_i,x_j) + f_{i,k}(x_i,x_k) + f_{j,k}(x_j,x_k) + f_{i,j,k}(x_i,x_j,x_k). \quad (7.78)\end{aligned}$$

The previous computations allows to get the decomposition functions.

$$\begin{aligned}f_0 &= \int_0^1 f(x)dx \quad (7.79)\\ f_i(x_i) &= -f_0 + \int_0^1 f(x)dx_{\sim i} \quad (7.80)\\ f_{i,j}(x_i,x_j) &= -f_0 - f_i(x_i) - f_j(x_j) + \int_0^1 f(x)dx_{\sim i,j}, \quad (7.81)\\ f_{i,j,k}(x_i,x_j,x_k) &= -f_0 - f_i(x_i) - f_j(x_j) - f_k(x_k) - f_{i,j}(x_i,x_j) - f_{i,k}(x_i,x_k) \quad (7.82)\\ & \quad - f_{j,k}(x_j,x_k) + \int_0^1 f(x)dx_{\sim i,j,k}, \quad (7.83)\end{aligned}$$

until the last term

$$\begin{aligned}f_{1,2,\ldots,p}(x_1,x_2,\ldots,x_p) &= f(x) - f_0 - \sum_{i=1,2,\ldots,p} f_i(x_i) - \ldots \quad (7.84)\\ & - \sum_{1\leq i_1<\ldots<i_{p-1}\leq p} f_{i_1,\ldots,i_{p-1}}(x_{i_1},\ldots,x_{i_{p-1}}). \quad (7.85)\end{aligned}$$

The last term is obviously so that the equality 7.70 is satisfied.

We have considered a function where the variables are in $[0,1]^p$. In fact, when we consider the more general model $Y = f(X_1,\ldots,X_p)$ where the random variables X_i are independent and uniform in $[0,1]^p$, the decomposition 7.70 is still valid.

7.8 Decomposition of the expectation

We can consider the decomposition 7.70 in terms of expectation and variance.

If we compute the expectation of Y by the expression 7.70 we get

$$E(Y) = f_0, \quad (7.86)$$

which is an obvious consequence of the zero integral property 7.71.

We can compute the first order decomposition functions f_i, by computing the conditional expectation with respect to X_i. Indeed, since the conditional expectation with respect to X_i is

$$E(Y|X_i) = \int_0^1 f(x)dx_{\sim i}, \quad (7.87)$$

for all $i = 1,2,\ldots,p$, the equation 7.80 can be written as

$$f_i(x_i) = -f_0 + E(Y|X_i). \quad (7.88)$$

We now plug the equation 7.86 into the previous equality, and get

$$f_i(x_i) \quad = \quad E(Y|X_i) - E(Y). \tag{7.89}$$

Similarily, we can compute the first order decomposition functions $f_{i,j}$, by computing the conditional expectation with respect to X_i and X_j. Indeed, since the conditional expectation with respect to X_i is

$$E(Y|X_i, X_j) \quad = \quad \int_0^1 f(x) dx_{\sim i,j}, \tag{7.90}$$

for all $i = 1, 2, \ldots, p$, the equation 7.81 can be written as

$$\begin{aligned} f_{i,j}(x_i, x_j) &= -f_0 - f_i(x_i) - f_j(x_j) + E(Y|X_i, X_j), && (7.91)\\ &= E(Y|X_i, X_j) - E(Y) - E(Y|X_i) - E(Y|X_j). && (7.92) \end{aligned}$$

Similarily, the equation 7.83 leads to

$$\begin{aligned} f_{i,j,k}(x_i, x_j, x_k) &= -f_0 - f_i(x_i) - f_j(x_j) - f_k(x_k) - f_{i,j}(x_i, x_j) - f_{i,k}(x_i, x_k) && (7.93)\\ &\quad -f_{j,k}(x_j, x_k) + E(Y|X_i, X_j, X_k). && (7.94)\\ &= E(Y|X_i, X_j, X_k) - E(Y) - E(Y|X_i) - E(Y|X_j) - E(Y|X_k) && (7.95)\\ &\quad -E(Y|X_i, X_j) - E(Y|X_i, X_k) - E(Y|X_j, X_k). && (7.96) \end{aligned}$$

TODO

7.9 Decomposition of the variance

TODO

7.10 Total sensitivity indices

TODO

7.11 Ishigami function

In this section, we consider the model

$$Y = f(X_1, X_2, X_3) = \sin(X_1) + a\sin^2(X_2) + bX_3^4 \sin(X_1) \tag{7.97}$$

where X_1, X_2, X_3 are three random variables uniform in $[-\pi, \pi]$. This implies that the distribution function of the variable X_i, f_i, satisfies the equation

$$f_i(X_i) = \frac{1}{2\pi}, \tag{7.98}$$

for $i = 1, 2, 3$.

We are going to compute the expectation, the variance and the sensitivity indices of this function. Before this, we need auxialiary results which are presented first.

7.11.1 Elementary integration

We first notice that the integral of the sin function in the interval $[-\pi, \pi]$ is zero, since this function is symetric. Hence,

$$\int_{-\pi}^{\pi} \sin(x)dx = 0. \tag{7.99}$$

We are going to prove that

$$\int_{-\pi}^{\pi} \sin^2(x)dx = \pi. \tag{7.100}$$

Indeed, if we integrate the $\sin^2(x)$ function by part, we get

$$\begin{aligned}\int_{-\pi}^{\pi} \sin^2(x)dx &= [-\cos(x)\sin(x)]_{-\pi}^{\pi} - \int_{-\pi}^{\pi} (-\cos(x))\cos(x)dx && (7.101)\\ &= 0 + \int_{-\pi}^{\pi} \cos^2(x)dx. && (7.102)\end{aligned}$$

On the other hand, the equality $\cos^2(x) + \sin^2(x) = 1$ implies

$$\begin{aligned}\int_{-\pi}^{\pi} \sin^2(x)dx &= \int_{-\pi}^{\pi} (1 - \cos^2(x))dx && (7.103)\\ &= 2\pi - \int_{-\pi}^{\pi} \cos^2(x)dx. && (7.104)\end{aligned}$$

We now combine 7.102 and 7.104 and get

$$\int_{-\pi}^{\pi} \cos^2(x)dx = 2\pi - \int_{-\pi}^{\pi} \cos^2(x)dx. \tag{7.105}$$

The previous equality implies that

$$2\int_{-\pi}^{\pi} \cos^2(x)dx = 2\pi, \tag{7.106}$$

which leads to

$$\int_{-\pi}^{\pi} \cos^2(x)dx = \pi. \tag{7.107}$$

Finally, the previous equality, combined with 7.102 immediately leads to 7.100.

We are going to prove that

$$\int_{-\pi}^{\pi} \sin^4(x)dx = \frac{3\pi}{4}. \tag{7.108}$$

Indeed, if we integrate the $\sin^4(x)$ function by part, we get

$$\begin{aligned}\int_{-\pi}^{\pi} \sin^4(x)dx &= \left[-\cos(x)\sin^3(x)\right]_{-\pi}^{\pi} - \int_{-\pi}^{\pi} (-\cos(x))(3\sin^2(x)\cos(x))dx && (7.109)\\ &= 0 + 3\int_{-\pi}^{\pi} \cos^2(x)\sin^2(x)dx. && (7.110)\end{aligned}$$

On the other hand, the equality $\cos^2(x) + \sin^2(x) = 1$ implies

$$\int_{-\pi}^{\pi} \sin^4(x)dx = \int_{-\pi}^{\pi} \sin^2(x)\sin^2(x)dx \quad (7.111)$$

$$= \int_{-\pi}^{\pi} (1 - \cos^2(x))\sin^2(x)dx \quad (7.112)$$

$$= \int_{-\pi}^{\pi} \sin^2(x)dx - \int_{-\pi}^{\pi} \cos^2(x)\sin^2(x)dx. \quad (7.113)$$

We plug the equality 7.100 into the previous equation and get

$$\int_{-\pi}^{\pi} \sin^4(x)dx = \pi - \int_{-\pi}^{\pi} \cos^2(x)\sin^2(x)dx. \quad (7.114)$$

We combine 7.110 and 7.110 and get

$$3\int_{-\pi}^{\pi} \cos^2(x)\sin^2(x)dx = \pi - \int_{-\pi}^{\pi} \cos^2(x)\sin^2(x)dx. \quad (7.115)$$

The previous equation leads to

$$4\int_{-\pi}^{\pi} \cos^2(x)\sin^2(x)dx = \pi, \quad (7.116)$$

which implies

$$\int_{-\pi}^{\pi} \cos^2(x)\sin^2(x)dx = \frac{\pi}{4}, \quad (7.117)$$

We finally plug the equation 7.117 into 7.110 and get the equation 7.108.

7.11.2 Expectation

By assumption, the three random variables X_1, X_2 and X_3 are independent, so that the joint distribution function is the product of the three distribution functions f_i, i.e.

$$g_{1,2,3}(x_1, x_2, x_3) = g_1(x_1)g_2(x_2)g_3(x_3). \quad (7.118)$$

By definition, the expectation of the random variable $\sin(X_1)$ is

$$E(\sin(X_1)) = \int_{-\pi}^{\pi}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} \sin(x_1)g_{1,2,3}(x_1, x_2, x_3)dx_1dx_2dx_3 \quad (7.119)$$

$$= \int_{-\pi}^{\pi}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} \sin(x_1)g_1(x_1)g_2(x_2)g_3(x_3)dx_1dx_2dx_3 \quad (7.120)$$

$$= \frac{1}{2\pi}\int_{-\pi}^{\pi} \sin(x_1)dx_1 \quad (7.121)$$

$$= 0. \quad (7.122)$$

By definition, the expectation of the random variable $\sin^2(X_2)$ is

$$E(\sin^2(X_2)) = \int_{-\pi}^{\pi}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} \sin^2(x_2) g_{1,2,3}(x_1,x_2,x_3) dx_1 dx_2 dx_3 \quad (7.123)$$

$$= \int_{-\pi}^{\pi} \sin^2(x_2) g_2(x_2) dx_2 \quad (7.124)$$

$$= \frac{1}{2\pi}\int_{-\pi}^{\pi} \sin^2(x_2) dx_2. \quad (7.125)$$

The equality 7.100 then implies that

$$E(\sin^2(X_2)) = \frac{1}{2\pi}\cdot\pi \quad (7.126)$$

$$= \frac{1}{2}. \quad (7.127)$$

By definition, the expectation of the random variable X_3^4 is

$$E(X_3^4) = \int_{-\pi}^{\pi}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} x_3^4 g_{1,2,3}(x_1,x_2,x_3) dx_1 dx_2 dx_3 \quad (7.128)$$

$$= \int_{-\pi}^{\pi}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} x_3^4 g_1(x_1) g_2(x_2) g_3(x_3) dx_1 dx_2 dx_3 \quad (7.129)$$

$$= \int_{-\pi}^{\pi} x_3^4 g_3(x_3) dx_3 \quad (7.130)$$

$$= \frac{1}{2\pi}\int_{-\pi}^{\pi} x_3^4 dx_3, \quad (7.131)$$

$$= \frac{1}{2\pi}\left[\frac{1}{5}x_3^5\right]_{-\pi}^{\pi} \quad (7.132)$$

$$= \frac{1}{2\pi}\left(\frac{1}{5}\pi^5 - \frac{1}{5}(-\pi)^5\right) \quad (7.133)$$

$$= \frac{1}{2\pi}\frac{2}{5}\pi^5 \quad (7.134)$$

$$= \frac{1}{5}\pi^4. \quad (7.135)$$

We are now going to use the expectations 7.122, 7.127 and 7.135 in order to compute the expectation of the output Y. The model 7.97 is a sum of functions. Since the expectation of a sum of two random variables is the sum of the expectations (be the variables independent or not), we have

$$E(Y) = E(\sin(X_1)) + E(a\sin^2(X_1)) + E(bX_3^4\sin(X_1)) \quad (7.136)$$

$$= E(\sin(X_1)) + aE(\sin^2(X_1)) + bE(X_3^4\sin(X_1)). \quad (7.137)$$

The expectation of the variable $X_3^4\sin(X_1)$ is

$$E(X_3^4\sin(X_1)) = \int_{-\pi}^{\pi}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} (x_3^4\sin(x_1)) g_{1,2,3}(x_1,x_2,x_3) dx_1 dx_2 dx_3 \quad (7.138)$$

$$= \int_{-\pi}^{\pi}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} (x_3^4\sin(x_1)) g_1(x_1) g_2(x_2) g_3(x_3) dx_1 dx_2 dx_3 \quad (7.139)$$

$$= E(X_3^4)E(\sin(X_1)). \quad (7.140)$$

Hence, the expectation of Y is

$$E(Y) = E(\sin(X_1)) + aE(\sin^2(X_1)) + bE(X_3^4)E(\sin(X_1)). \quad (7.141)$$

We now combine the equations 7.122, 7.127 and 7.135 and get

$$E(Y) = 0 + a\frac{1}{2} + b\frac{1}{5}\pi^4 \cdot 0 \quad (7.142)$$

$$= \frac{a}{2}. \quad (7.143)$$

7.11.3 Variance

The variance of the output Y is

$$V(Y) = E(Y^2) - E(Y)^2 \quad (7.144)$$

$$= E\left(\left(\sin(X_1) + a\sin^2(X_2) + bX_3^4\sin(X_1)\right)^2\right) - E(Y)^2 \quad (7.145)$$

$$= E\left(\sin^2(X_1) + a^2\sin^4(X_2) + b^2X_3^8\sin^2(X_1) + \right. \quad (7.146)$$

$$2\sin(X_1)a\sin^2(X_2) + 2\sin^2(X_1)bX_3^4 + \quad (7.147)$$

$$\left. 2a\sin^2(X_2)bX_3^4\sin(X_1)\right) - E(Y)^2 \quad (7.148)$$

$$= E(\sin^2(X_1)) + a^2E(\sin^4(X_2)) + b^2E(X_3^8)E(\sin^2(X_1)) + \quad (7.149)$$

$$2aE(\sin(X_1))E(\sin^2(X_2)) + 2bE(\sin^2(X_1))E(X_3^4) + \quad (7.150)$$

$$2abE(\sin^2(X_2))E(X_3^4)E(\sin(X_1)) - E(Y)^2. \quad (7.151)$$

By the equality 7.122, the expectation of $\sin(X_1)$ is zero in the interval $[-\pi, \pi]$. Therefore, the terms associated with $E(\sin(X_1))$ can be simplified in the previous equality. This leads to

$$V(Y) = E(\sin^2(X_1)) + a^2E(\sin^4(X_2)) + b^2E(X_3^8)E(\sin^2(X_1)) + \quad (7.152)$$

$$2bE(X_3^4)E(\sin^2(X_1)) - E(Y)^2 \quad (7.153)$$

We now compute the terms appearing in the previous equality. Actually, we do not have much to compute, since the equalities 7.127 and 7.135 are already available. Indeed, the equality 7.127 immediately leads to

$$E(\sin^2(X_1)) = E(\sin^2(X_2)) \quad (7.154)$$

$$= \frac{1}{2}. \quad (7.155)$$

What remains to compute is $E(\sin^4(X_2))$ and $E(X_3^8)$.

By definition, the expectation of the random variable X_3^4 is

$$E(X_3^8) = \frac{1}{2\pi}\int_{-\pi}^{\pi} x_3^8 dx_3, \quad (7.156)$$

$$= \frac{1}{2\pi}\left[\frac{1}{9}x_3^9\right]_{-\pi}^{\pi} \quad (7.157)$$

$$= \frac{1}{2\pi}\left(\frac{1}{9}\pi^9 - \frac{1}{9}(-\pi)^9\right) \quad (7.158)$$

$$= \frac{1}{2\pi}\frac{2}{9}\pi^9 \quad (7.159)$$

$$= \frac{1}{9}\pi^8. \quad (7.160)$$

On the other hand, the expectation of the random variable $\sin^4(X_2)$ is

$$E(\sin^2(X_2)) = \frac{1}{2\pi}\int_{-\pi}^{\pi} \sin^4(x_2)dx_2. \tag{7.161}$$

The equality 7.108 then implies that

$$\begin{aligned} E(\sin^4(X_2)) &= \frac{1}{2\pi}\cdot\frac{3\pi}{4} && (7.162)\\ &= \frac{3}{8}. && (7.163)\end{aligned}$$

We now plug the equalities 7.155, 7.135, 7.160 and 7.163 into 7.153, and get

$$\begin{aligned} V(Y) &= \frac{1}{2} + a^2\frac{3}{8} + b^2\frac{1}{9}\pi^8\frac{1}{2} + 2b\frac{1}{5}\pi^4\frac{1}{2} - \frac{a^2}{2^2} && (7.164)\\ &= \frac{1}{2} + \frac{3a^2}{8} + \frac{b^2\pi^8}{18} + \frac{b\pi^4}{5} - \frac{a^2}{4} && (7.165)\\ &= \frac{1}{2} + \frac{a^2}{8} + \frac{b^2\pi^8}{18} + \frac{b\pi^4}{5} && (7.166)\end{aligned}$$

7.11.4 Sobol decomposition

In this section, we perform the Sobol decompotion of the function f, as presented in the section 7.7.

By the equation 7.80, we have

$$f_1(X_1) = -f_0 + \int_0^1 f(x)dx_{\sim 1} \tag{7.167}$$

TODO

7.11.5 The Sobol method for sensitivity analysis

TODO

7.11.6 The Ishigami function by the Sobol method

In this section, we compute the sensitivity indices of the Ishigami function by the Sobol method. We consider three random variables uniform in $[-\pi, \pi]$. We use Monte-Carlo experiments to compute the sensitivity indices.

The following script allows to perform the analysis.

```
function y = ishigami (x)
  a=7.
  b=0.1
  s1=sin(x(:,1))
  s2=sin(x(:,2))
  x34 = x(:,3).^4
```

```
  y(:,1) = s1 + a.*s2.^2 + b.*x34.*s1
endfunction

function C = nisp_cov ( x , y )
  // Returns the empirical covariance matrix of x and y.
  x=x(:)
  y=y(:)
  n = size(x,"*")
  x=x-mean(x)
  y=y-mean(y)
  C(1,1) = x'*x/(n-1)
  C(1,2) = x'*y/(n-1)
  C(2,1) = C(1,2)
  C(2,2) = y'*y/(n-1)
endfunction

function s = sensitivityindex(ya,yc)
  // Returns the sensitivity index
  // associated with experiments ya and yc.
  C = nisp_cov (ya, yc)
  s= C(1,2)/ (st_deviation(ya) * st_deviation(yc))
endfunction

// Create the uncertain parameters
rvu1 = randvar_new("Uniforme",-%pi,%pi);
rvu2 = randvar_new("Uniforme",-%pi,%pi);
rvu3 = randvar_new("Uniforme",-%pi,%pi);
srvu = setrandvar_new();
setrandvar_addrandvar ( srvu, rvu1);
setrandvar_addrandvar ( srvu, rvu2);
setrandvar_addrandvar ( srvu, rvu3);
// The number of uncertain parameters is :
nx = setrandvar_getdimension(srvu);
np = 10000;
// Create a first sampling A
setrandvar_buildsample(srvu,"Lhs",np);
A = setrandvar_getsample(srvu);
// Create a first sampling B
setrandvar_buildsample(srvu,"Lhs",np);
B = setrandvar_getsample(srvu);
// Perform the experiments in A
ya = ishigami(A);
// Compute the first order sensitivity index for X1
C = B;
C(1:np,1)=A(1:np,1);
yc = ishigami(C);
s1 = sensitivityindex(ya,yc);
mprintf("S1 : %f (expected = %f)\n", s1, 0.3139);
// Compute the first order sensitivity index for X2
```

```
C = B;
C(1:np,2)=A(1:np,2);
yc = ishigami(C);
s2 = sensitivityindex(ya,yc);
mprintf("S2 : %f (expected = %f)\n", s2, 0.4424);
// Compute the first order sensitivity index for X3
C = B;
C(1:np,3)=A(1:np,3);
yc = ishigami(C);
s3 = sensitivityindex(ya,yc);
mprintf("S3 : %f (expected = %f)\n", s3, 0.0);
// Compute the first order sensitivity index for {X1,X3}
C = A;
C(1:np,2)=B(1:np,2);
yc = ishigami(C);
s13 = sensitivityindex(ya,yc);
mprintf("S13 : %f (expected = %f)\n", s13, 0.5576);
//
// Clean-up
randvar_destroy(rvu1);
randvar_destroy(rvu2);
randvar_destroy(rvu3);
setrandvar_destroy(srvu);
```

The previous script produces the following output.

```
S1 : 0.317135 (expected = 0.313900)
S2 : 0.449154 (expected = 0.442400)
S3 : 0.010450 (expected = 0.000000)
S13 : 0.561985 (expected = 0.557600)
```

In the following script, we create the histogram of the output of the Ishigami function.

```
scf();
histplot(50,ya)
xtitle("Ishigami function","X","P(X)")
```

The previous script produces the figure 7.6.

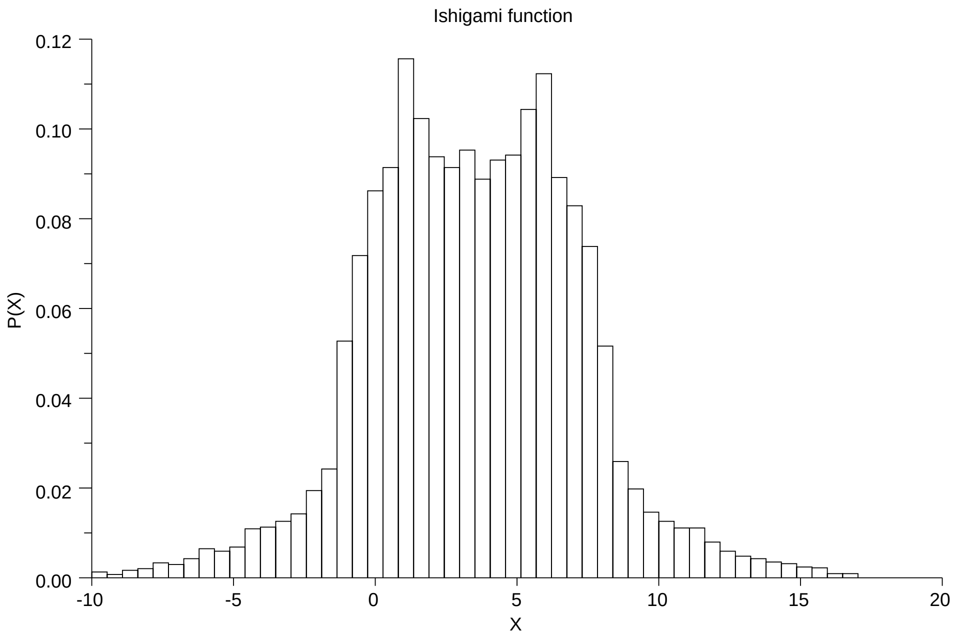

Figure 7.6: Histogram of the output of the Ishigami function.

Chapter 8

Thanks

Many thanks to Allan Cornet.

Bibliography

[1] T. Homma and A. Saltelli. Importance measures in global sensitivity analysis of non linear models. *Reliability Engineering and System Safety*, 52:1?–17, 1996.

[2] Julien Jacques. Contributions à l?analyse de sensibilité et à l?analyse discriminante généralisée, 2005.

[3] Didier Pelat. *Bases et méthodes pour le traitement des données (Bruits et Signaux).* Master M2 Recherche : Astronomie?astrophysique, 2006.

[4] I.M. Sobol. Sensitivity estimates for nonlinear mathematical models. *Mathematical Modelling and Computational Experiments*, 1:407?–414, 2003.