NISP Toolbox Manual

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

Version 0.2
January 2011

Abstract

This document is an introduction to the NISP module. We present the installation process of the module in binary from ATOMS or from the sources. Then we present the configuration functions and the randvar, setrandvar and polychaos classes. Several examples are provided for each class, which provides an overview of the use of NISP in practical situations. In the last section, we present an introduction to sensivity analysis and show how to use Scilab and the NISP module in this context.

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.

This module was presented in the "42ÃÍmes JournÃl'es de Statistique, du 24 au 28 mai 2010" [3].

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.

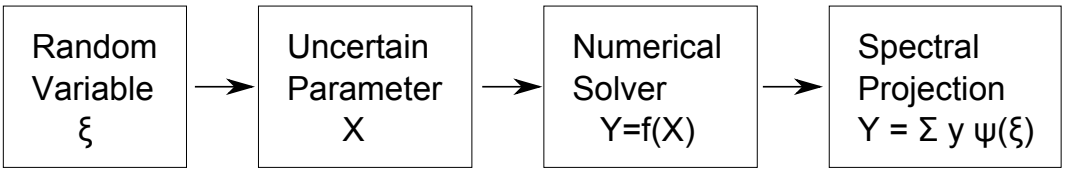

Figure 1.1: The NISP methodology

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.

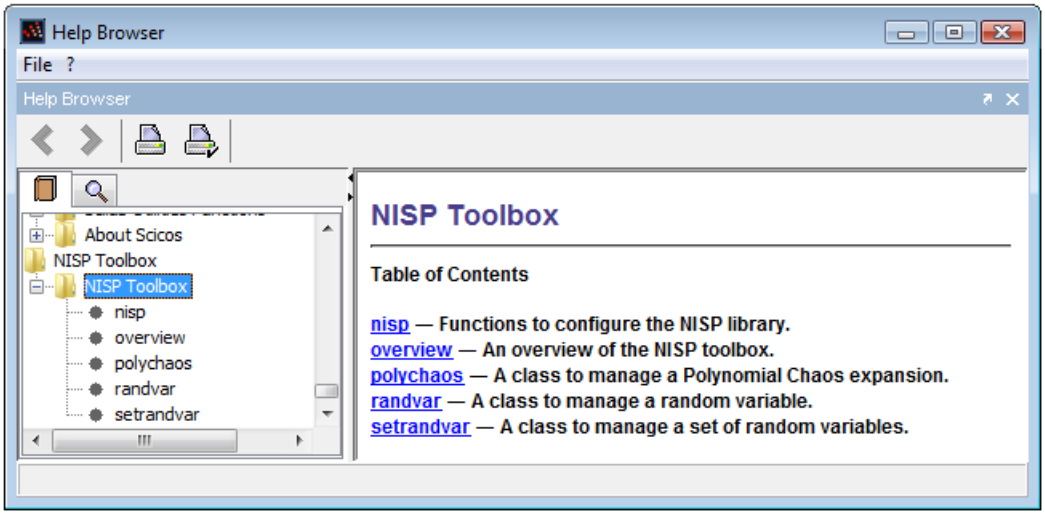

Figure 1.2: The NISP inline help.

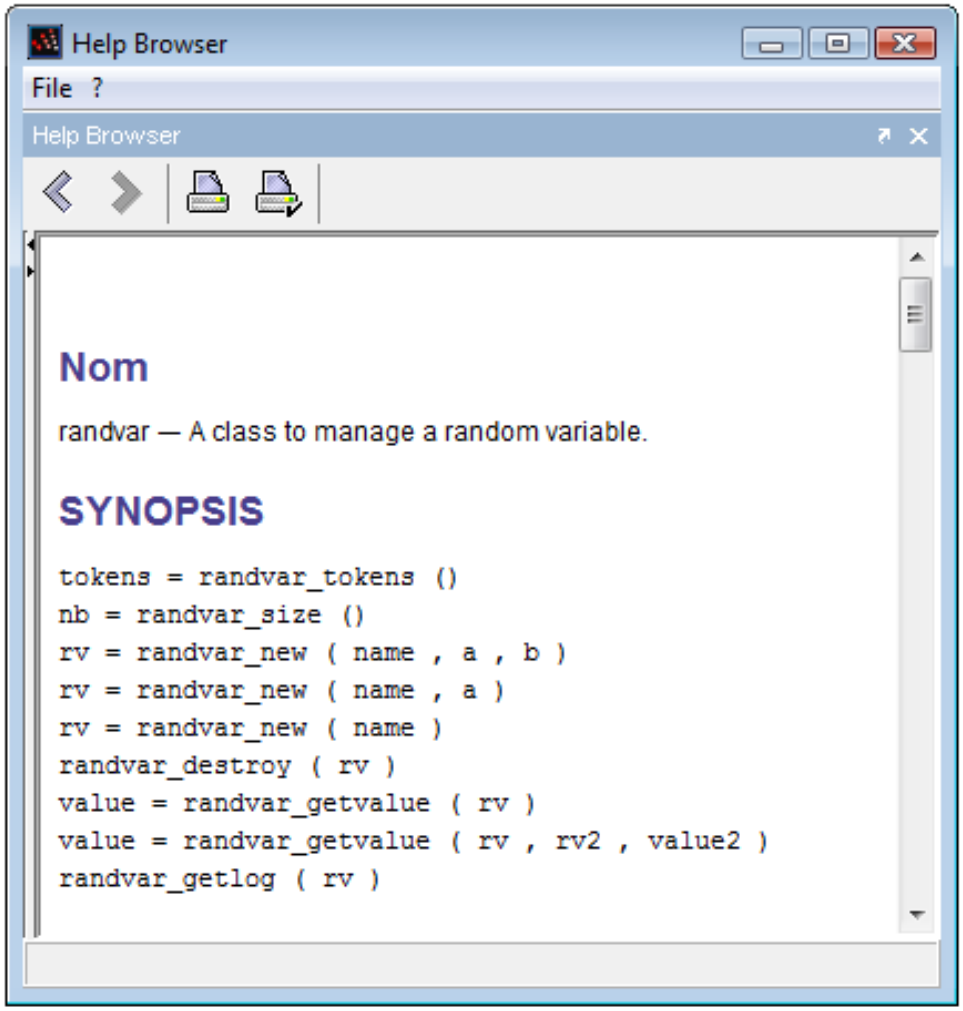

Figure 1.3: The online help of the randvar function.

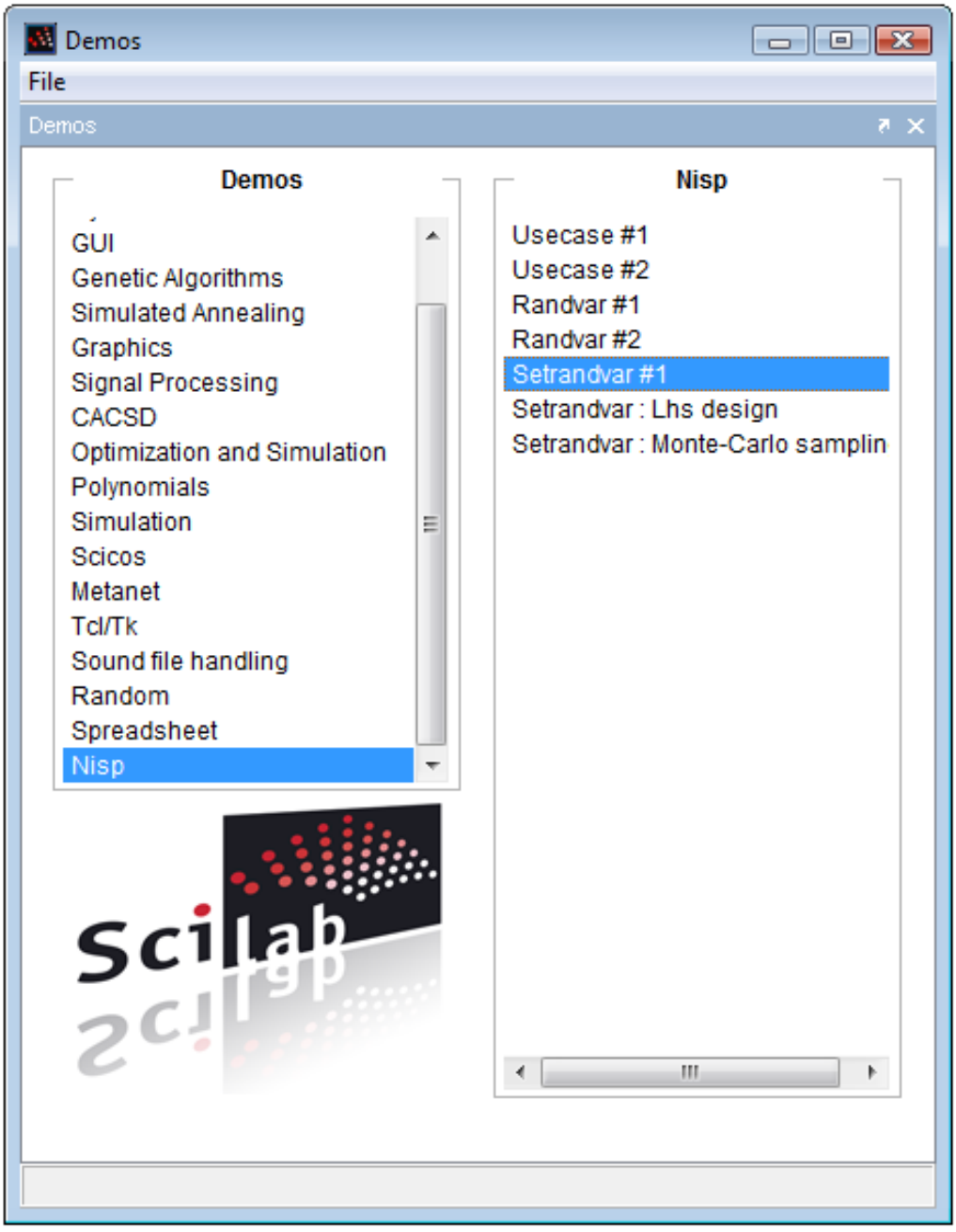

Figure 1.4: Demonstrations provided with the NISP toolbox.

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 [7].

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

A log-normal distribution is a probability distribution of a random variable whose logarithm is normally distributed. If X is a random variable with a normal distribution, then $Y = exp(X)$ has a 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

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

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

In the figure 4.2, we have $\mu' = E(X)$.

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

$$RAND\_MAX = 2^{15} - 1 = 32767. \tag{4.7}$$

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

$$\overline{X} = \frac{n}{N}, \tag{4.8}$$

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.

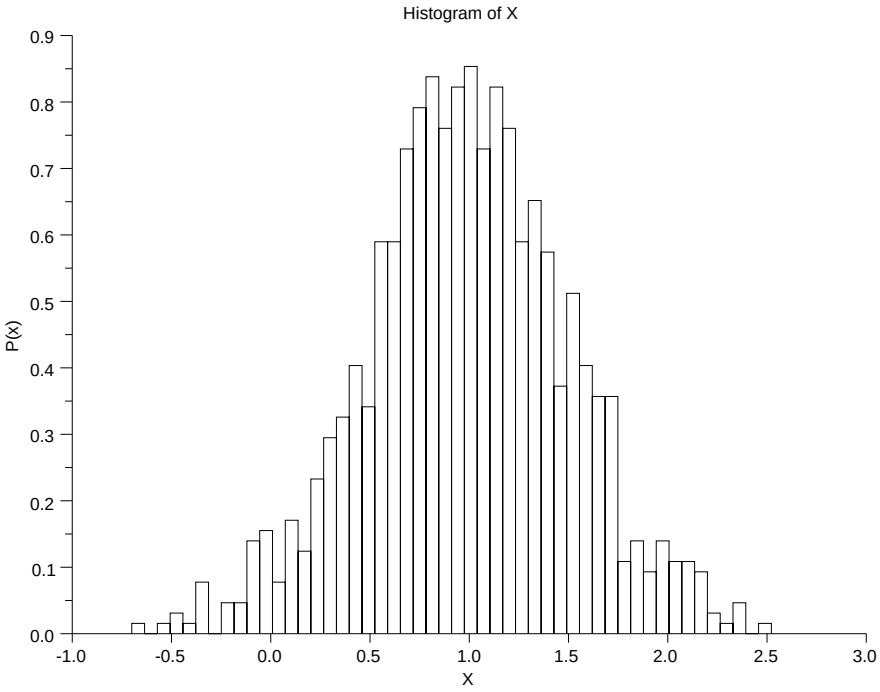

Figure 4.4: The histogram of a Normal random variable with 1000 samples.

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.

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

```
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.

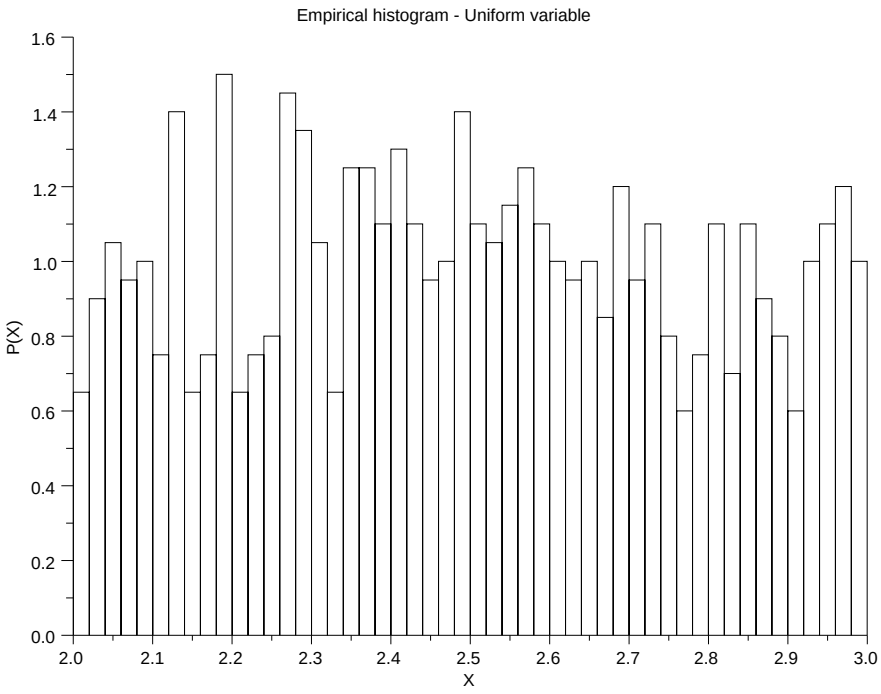

Figure 4.6: The histogram of a Uniform random variable with 1000 samples.

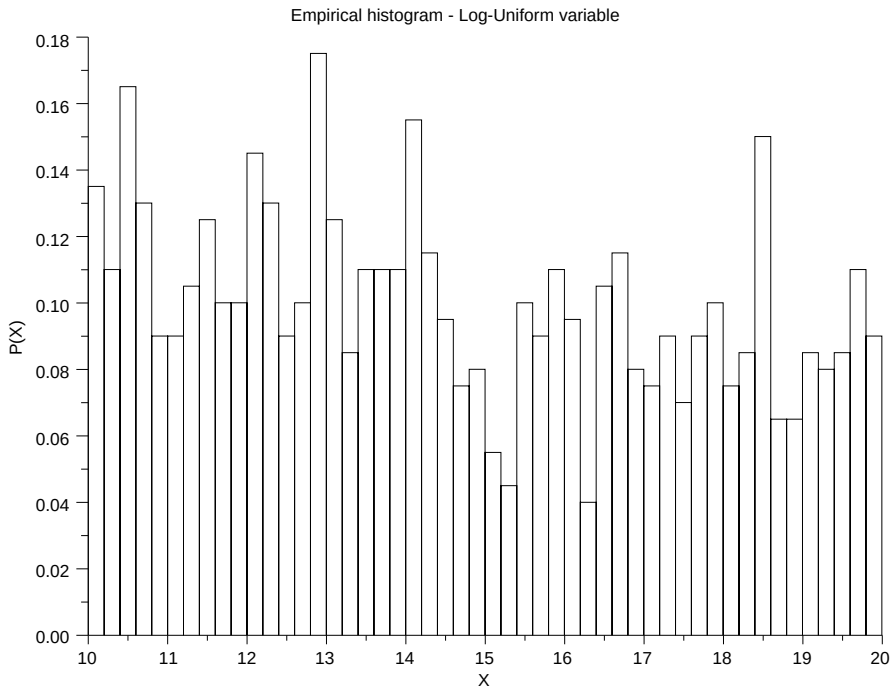

Figure 4.7: The histogram of a Log-Uniform random variable with 1000 samples.

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.

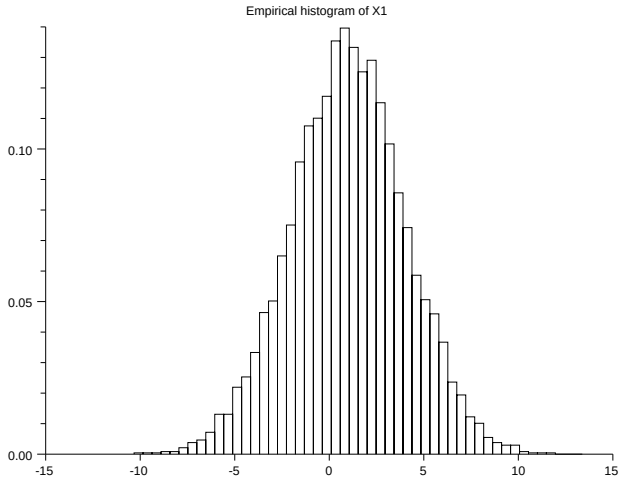

Figure 5.2: Monte-Carlo Sampling - Normal random variable.

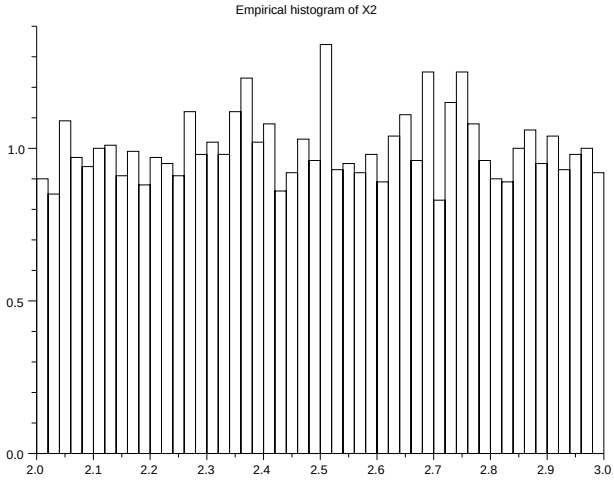

Figure 5.3: Monte-Carlo Sampling - Uniform random variable.

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.

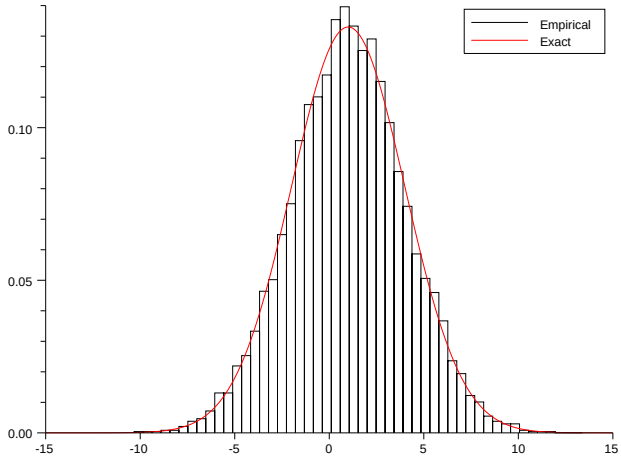

Figure 5.4: Monte-Carlo Sampling - Histogram and exact distribution functions for the first variable.

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.

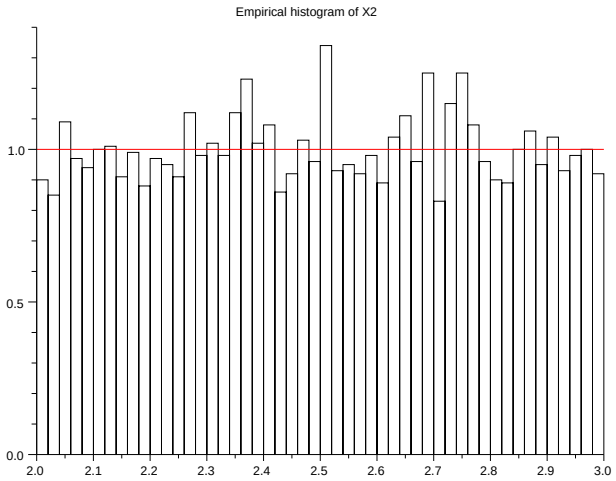

Figure 5.5: Monte-Carlo Sampling - Histogram and exact distribution functions for the second variable.

```
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.

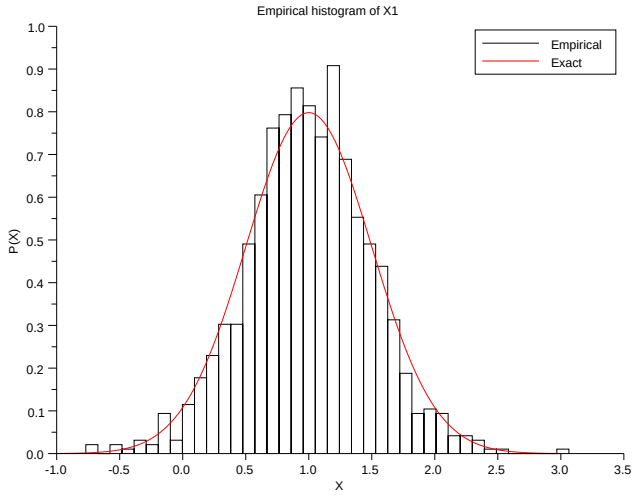

Figure 5.6: Monte-Carlo Sampling - Histogram and exact distribution functions for the first variable.

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.

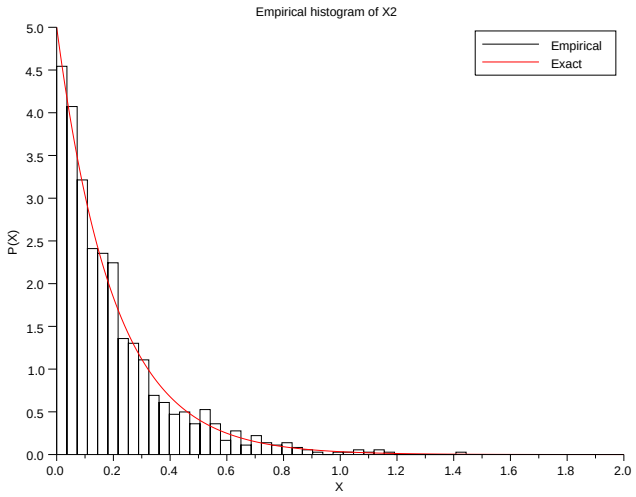

Figure 5.7: Monte-Carlo Sampling - Histogram and exact distribution functions for the second variable.

```
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
```

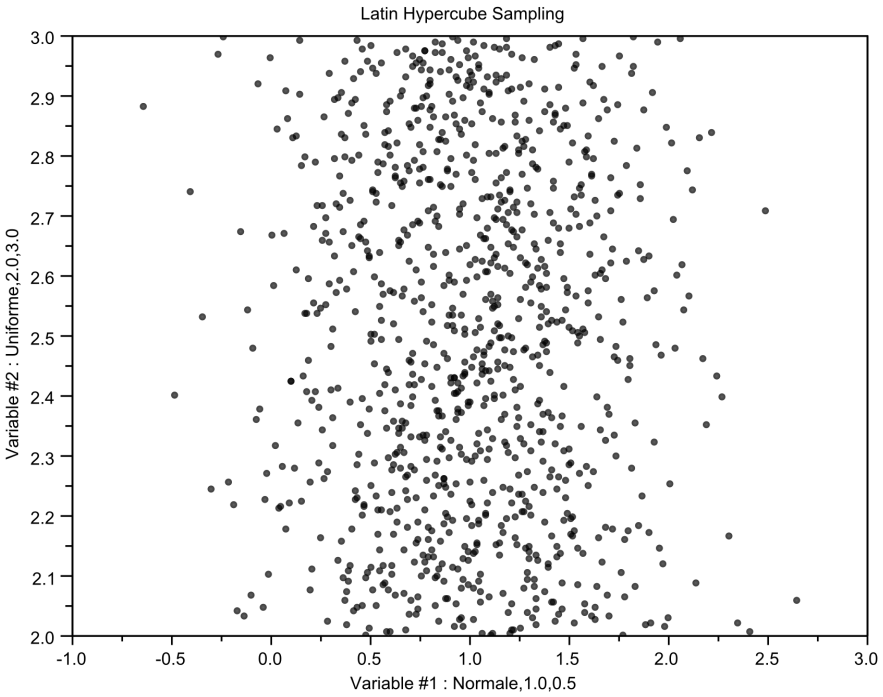

Figure 5.8: Latin Hypercube Sampling - The first variable is Normal, the second variable is Uniform.

```
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";
```

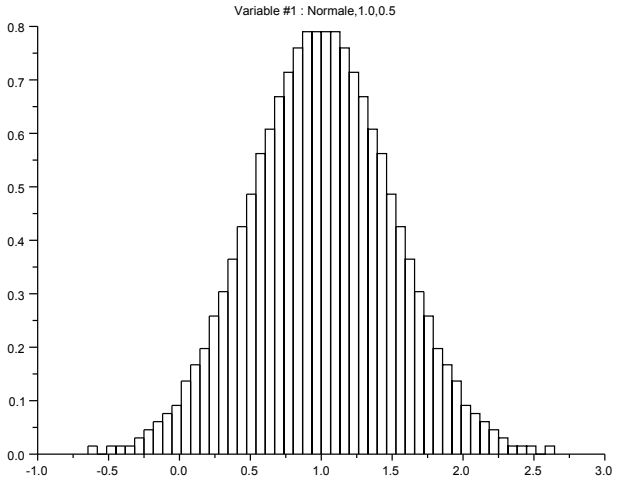

Figure 5.9: Latin Hypercube Sampling - Normal random variable.

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.

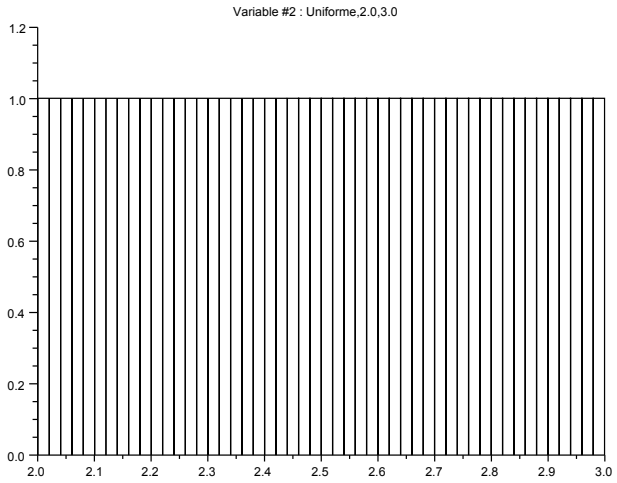

Figure 5.10: Latin Hypercube Sampling - Uniform random variable.

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

5.2.4 A note on the LHS samplings

We emphasize that the LHS sampling which is provided by the `setrandvar_buildsample` function is so that the points are centered within their cells.

In the following script, we create a LHS sampling with 10 points.

```
srv = setrandvar_new(2);
np = 10;
setrandvar_buildsample ( srv , "Lhs" , np );
sampling = setrandvar_getsample ( srv );
scf();
plot(sampling(:,1),sampling(:,2),"bo");
xtitle("LHS␣Design","X1","X2");
// Add the cuts
cut = linspace ( 0 , 1 , np + 1 );
for i = 1 : np + 1
  plot( [cut(i) cut(i)] , [0 1] , "-" )
end
for i = 1 : np + 1
  plot( [0 1] , [cut(i) cut(i)] , "-" )
end
setrandvar_destroy ( srv )
```

The previous script produces the figure 5.11.

The "LhsMaxMin" sampling provided by the `setrandvar_buildsample` function tries to maximize the minimum distance between the points in the sampling. The `ntry` parameter is the

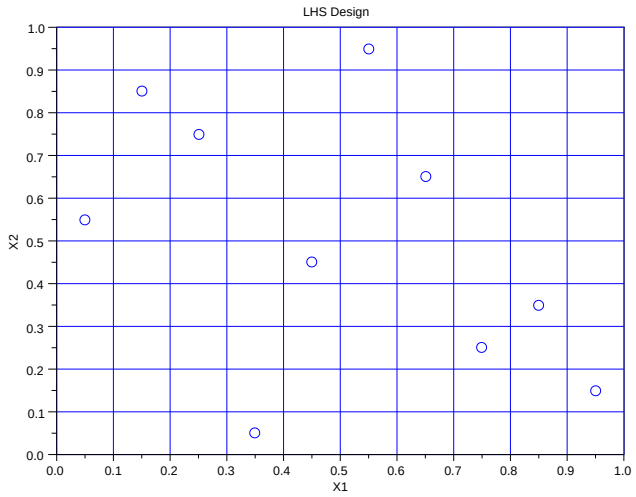

Figure 5.11: Latin Hypercube Sampling - Computed with `setrandvar_buildsample` and the "Lhs" option.

number of random points generated before the best is accepted in the sampling.

```
np = 10;
ntry = 100;
setrandvar_buildsample ( srv , "LhsMaxMin" , np , ntry );
sampling = setrandvar_getsample ( srv );
```

The previous script produces the figure 5.12.

On the other hand, the `nisp_buildlhs` function produces a more classical LHS sampling, where the points are randomly picked within their cells.

```
n = 5;
s = 2;
sampling = nisp_buildlhs ( s , n );
scf();
plot ( sampling(:,1) , sampling(:,2) , "bo" );
// Add the cuts
cut = linspace ( 0 , 1 , n + 1 );
for i = 1 : n + 1
plot( [cut(i) cut(i)] , [0 1] , "-" )
end
for i = 1 : n + 1
plot( [0 1] , [cut(i) cut(i)] , "-" )
end
```

The previous script produces the figure 5.13.

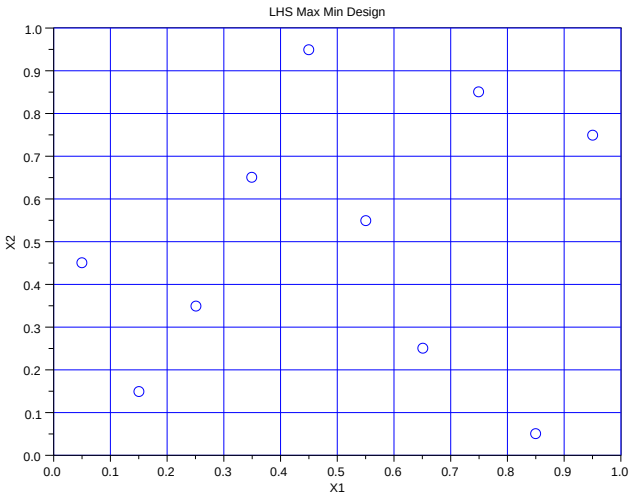

Figure 5.12: Latin Hypercube Sampling - Computed with `setrandvar_buildsample` and the "LhsMaxMin" option.

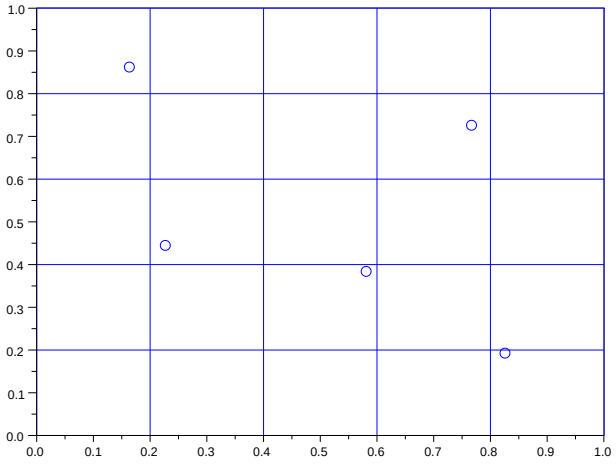

Figure 5.13: Latin Hypercube Sampling - Computed with `nisp_buildlhs`.

5.2.5 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 5.14 presents this sampling and the figures 5.15 and 5.16 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);
```

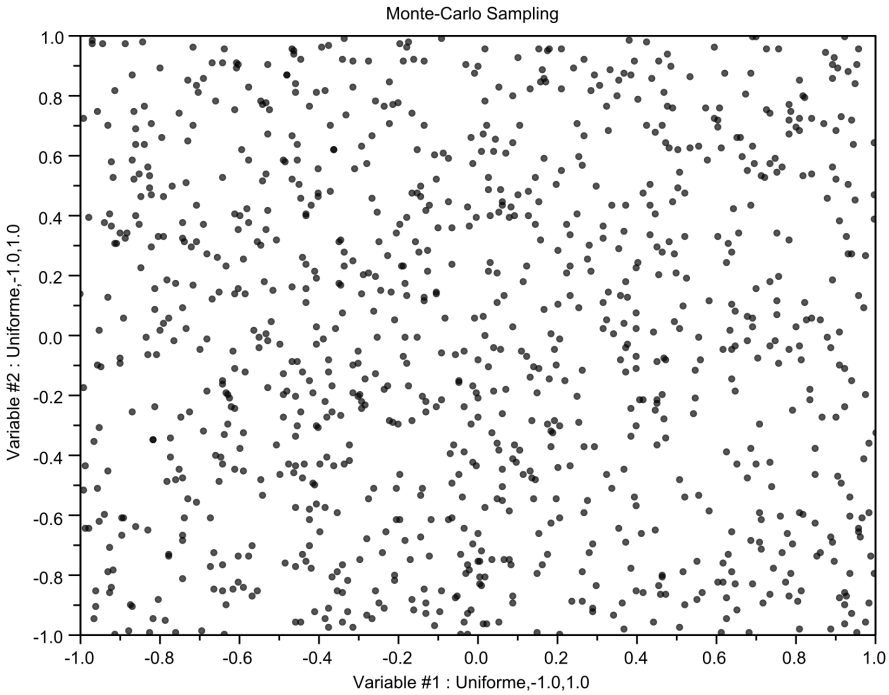

Figure 5.14: Monte-Carlo Sampling - Two uniform variables in the interval $[-1, 1]$.

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 5.17, 5.18 and 5.19.

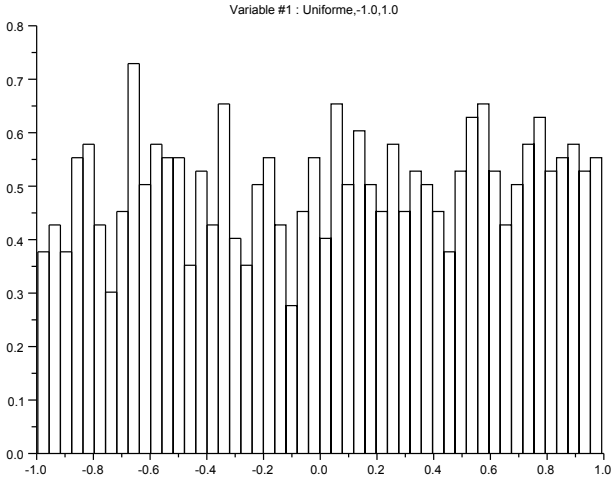

Figure 5.15: Latin Hypercube Sampling - First uniform variable in $[-1, 1]$.

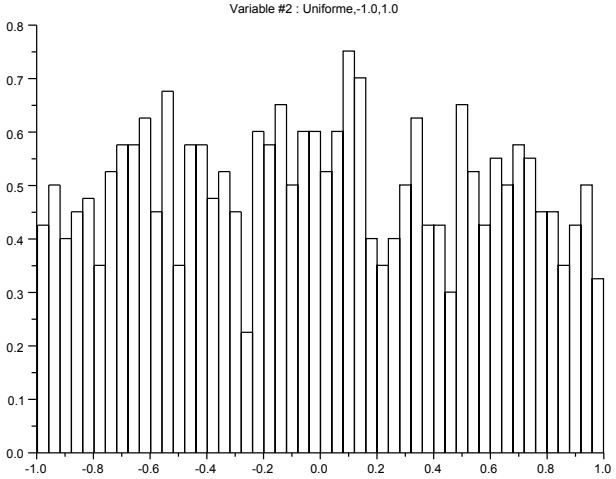

Figure 5.16: Latin Hypercube Sampling - Second uniform variable in $[-1, 1]$.

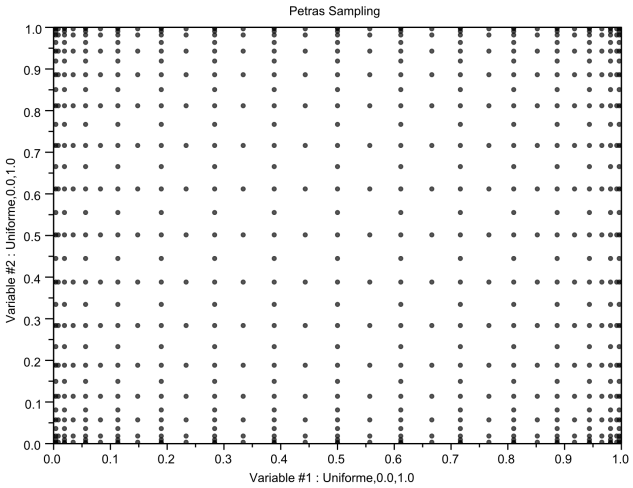

Figure 5.17: Petras sampling - Two uniform variables in the interval $[-1, 1]$.

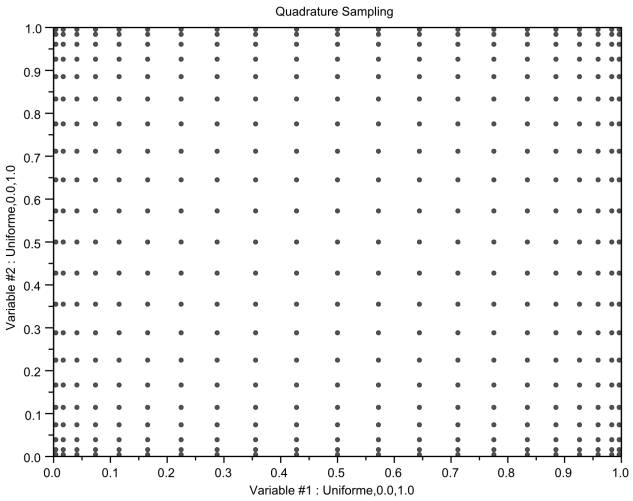

Figure 5.18: Quadrature sampling - Two uniform variables in the interval $[-1, 1]$.

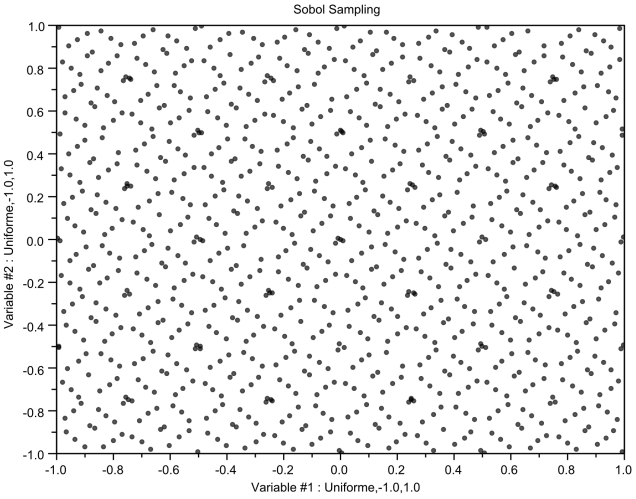

Figure 5.19: Sobol sampling - Two uniform variables in the interval $[-1, 1]$.

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.

```
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.

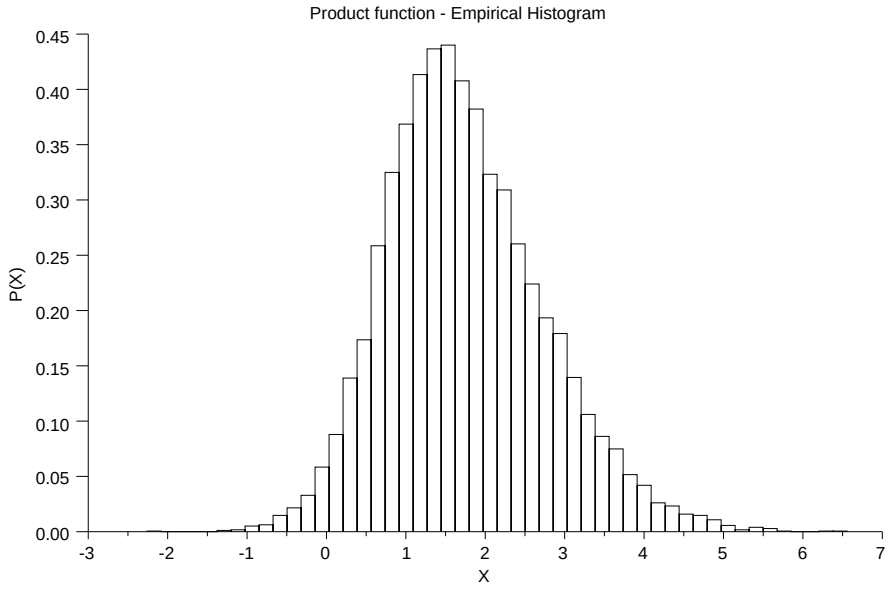

Figure 6.3: Product function - Histogram of the output on a LHS design with 10000 experiments.

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 A note on performance

In this section, we emphasize vectorization which can be used to improve the performance of a script when we compute the output of a function on a given sampling.

In order to use vectorization, the core feature that we used in the `Exemple` is the use of the elementwise multiplication, denoted by `.*`. In the Exemple function below (reproduced here for simplicity), the input `x` is a np-by-2 matrix of doubles, where np is the number of experiments, and y is a np-by-1 matrix of doubles.

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

The elementwise multiplication allows to multiply the two first columns of `x`, and sets the result into the output `y`, in one single statement. Since Scilab uses optimized numerical libraries, this allows to get the best performance in most situations.

In the previous section, we have shown that we can compute the output of the `Exemple` function in one single call to the function.

```
outputdata = Exemple(inputdata);
```

This call allows to produce all the outputs as fast as possible and is the recommended method. The reason is that the previous script lets Scilab perform computations with large matrices.

In fact, there is another, slower, method to perform the same computation. We 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
```

While the previous script is perfectly correct, it can be very slow when the number of experiments is large. This is because the interpreter has to perform a large number of loops with matrices of small size. In general, this produces much slower script and should be avoided. More details on this topic are presented in [2].

6.2.3 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.

```
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);
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");
```

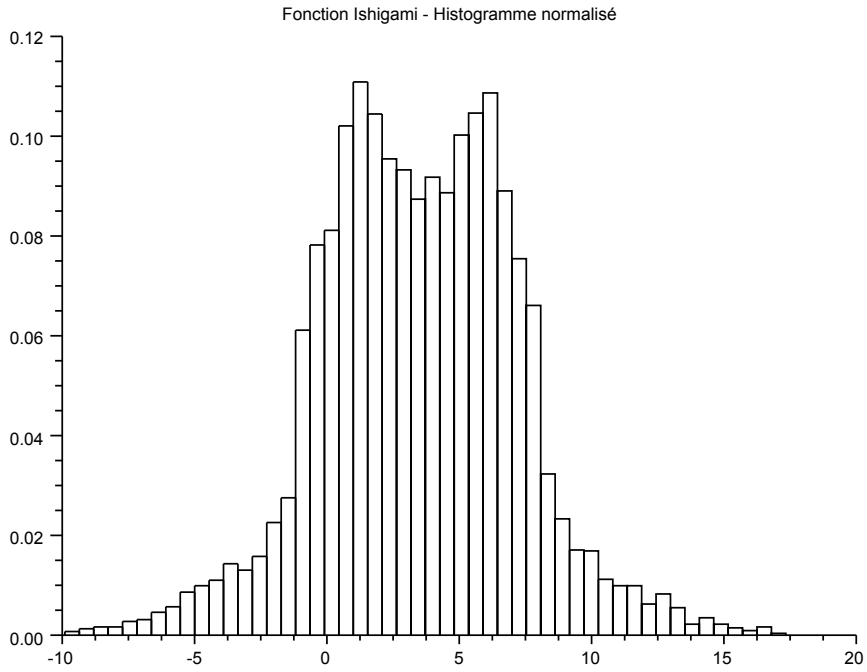

Figure 6.4: Ishigami function - Histogram of the output of the chaos polynomial on a LHS design with 10 000 experiments.

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();
```

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

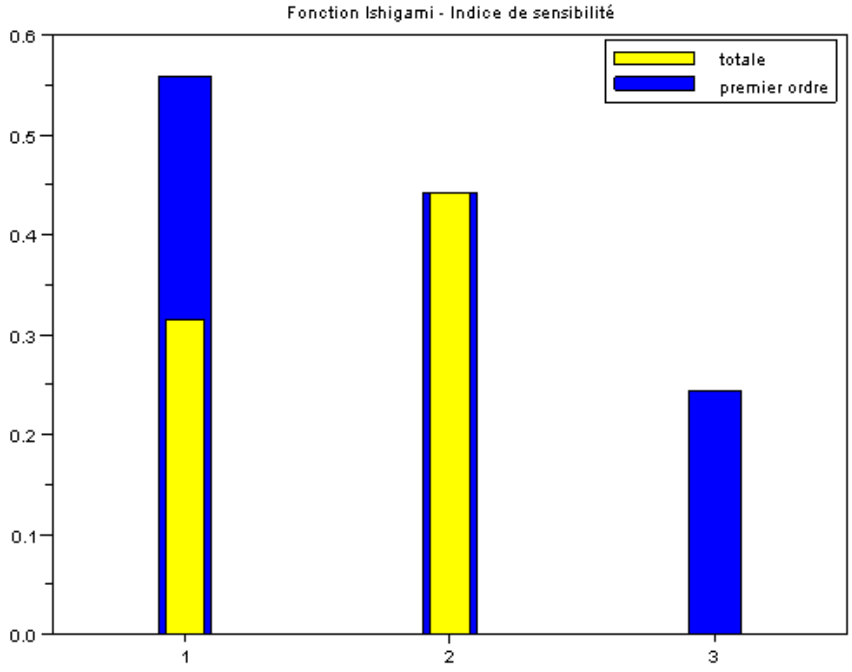

Figure 6.5: Ishigami function - Sensitivity indices.

Chapter 7

An 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.

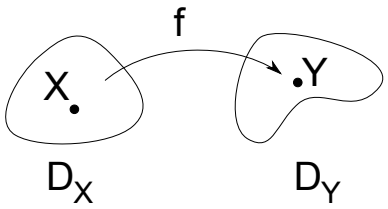

Figure 7.1: Global analysis.

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

$$V(Y) = \int_{D_X} (y - E(Y))^2 f_Y(y) dy. \tag{7.3}$$

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 [4] or in the work of Sobol [10]. The Thesis by Jacques [6] 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

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

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.

Definition 7.2.1. *(*Standardized Regression Coefficient*) The standardized regression coefficient is*

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

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

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

$$SRC_1 + SRC_2 + \ldots + SRC_p = 1. \tag{7.9}$$

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

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

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 An example of affine model

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.

Consider the 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. As we are going to see, all the input random variables do not have the same effect on the variability of the output Y.

We can immediately compute the expectation of the output Y. Since the input random variables are independent, we have

$$E(Y) = E(X_1) + E(X_2) + E(X_3) + E(X_4). \tag{7.22}$$

In our case, the input variables X_i have zero mean, so that the output random variable Y also has a zero mean.

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

$$V(X_1) = 1, \qquad V(X_2) = 4, \tag{7.23}$$

$$V(X_3) = 9, \qquad V(X_4) = 16. \tag{7.24}$$

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

$$V(Y) = V(X_1) + V(X_2) + V(X_3) + V(X_4) = 30. \tag{7.25}$$

The standardized regression coefficient is

$$SRC_i = \frac{\beta_i^2 V(X_i)}{V(Y)} = \frac{i^2}{30}, \tag{7.26}$$

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

$$SRC_1 = \frac{1}{30}, \qquad SRC_2 = \frac{4}{30}, \tag{7.27}$$

$$SRC_3 = \frac{9}{30}, \qquad SRC_4 = \frac{16}{30}. \tag{7.28}$$

We have the following inequalities:

$$SRC_4 > SRC_3 > SRC_2 > SRC_1. \tag{7.29}$$

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(sampling(:,k),y,'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 scatter plots for X_1, X_2, X_3 and X_4. The figure 7.2 present the scatter plot for X_4. The linearity of the function $Y = f(X)$ is obvious from these scatter plots.

The histogram of the output Y is presented in the figure 7.3. The symetric, bell-shaped curve

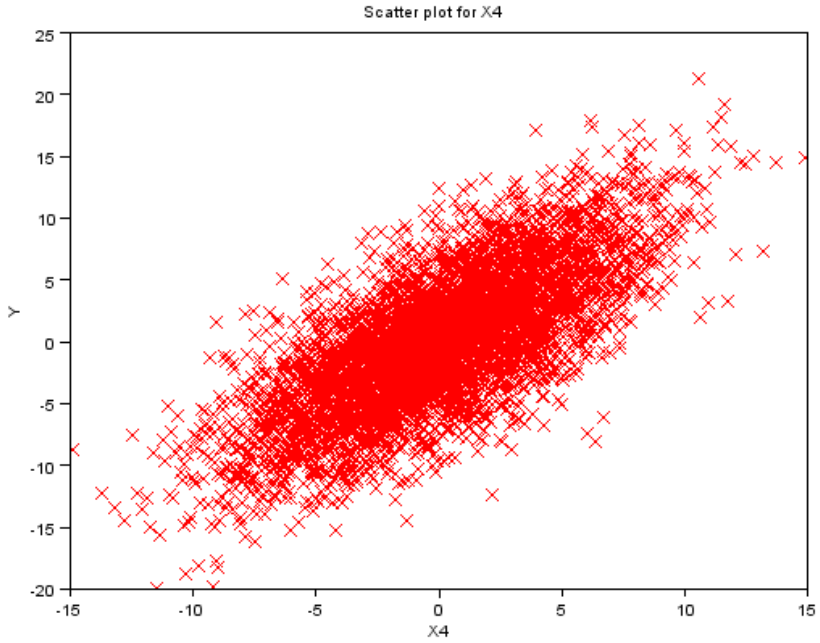

Figure 7.2: Scatter plot for an affine model - Variable X_4.

makes it clear that the output Y is a normal random variable. Indeed, we know from the previous analysis that the mean of Y is zero, and its variance is 30.

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 $\hat{x}_i$ for example, then the variance of the output Y may decrease or increase, depending on the value of $\hat{x}_i$, because the variable X_i is not random anymore. We can then measure the conditionnal variance given $X_i = \hat{x}_i$, denoted by $V(Y|X_i = \hat{x}_i)$.

Since X_i is a random variable, then the conditionnal variance $V(Y|X_i = \hat{x}_i)$ is a random variable. We have already emphasized that the variance $V(Y)$ can be smaller or larger than $V(Y|X_i = \hat{x}_i)$. The other problem is that, in most cases, we do not know where the value of X_i is best fixed. It may sound reasonable to investigate $V(Y|X_i = \hat{x}_i)$ when the random variable X_i has its mean value, i.e. it may be interesting to computed $V(Y|X_i = \overline{x}_i)$. But other values of X_i might change the variance significantly, so that the result may not be interesting.

It therefore sounds reasonable to average the measure $V(Y|X_i = \hat{x}_i)$ over all possible values of X_i. We are then 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

$$V(Y) = V(E(Y|X_i)) + E(V(Y|X_i)). \tag{7.30}$$

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

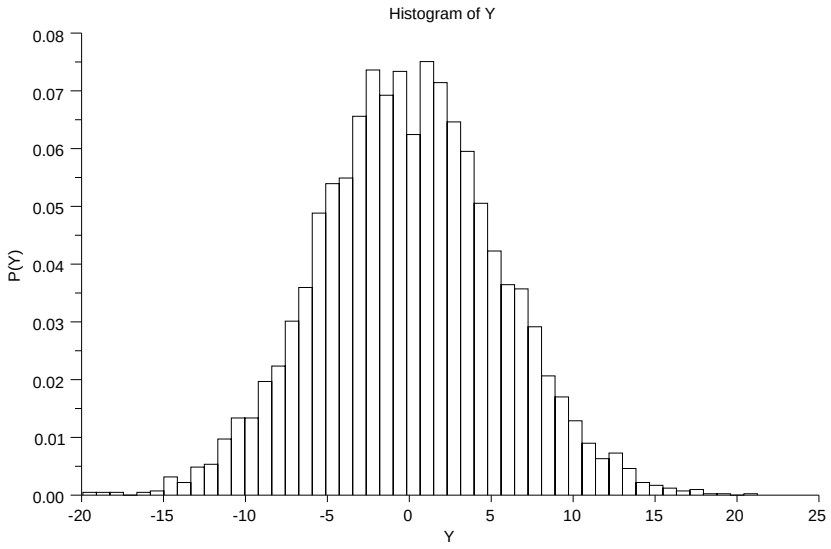

Figure 7.3: Histogram of the output Y for an affine model.

Definition 7.5.1. *(*First order sensitivity indices*) The first order sensivity indice of Y to the variable X_i is defined by*

$$S_i = \frac{V(E(Y|X_i))}{V(Y)}, \tag{7.31}$$

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

The sensitivity indice measures the part of the variance which is caused by the uncertainty in X_i.

In the following proposition, we compute the sensitivity indices when the function f is linear.

Proposition 7.5.2. *(*First order sensitivity indice of a linear model*) Assume that the output Y depends linearily on the input X_i:*

$$Y = \beta_0 + \sum_{i=1,2,\ldots,p} \beta_i X_i, \tag{7.32}$$

where $\beta_i \in \mathbb{R}$, for $i = 0, 1, 2, \ldots, p$. Assume that the input variables X_i are independent. Therefore, the sensitivity index of Y to the variable X_i is

$$S_i = \frac{\beta_i^2 V(X_i)}{V(Y)}, \tag{7.33}$$

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

Proof. We consider the expectation of the equation 7.32 and get:

$$E(Y|X_i) = \beta_0 + \sum_{i=1,2,\ldots,i-1,i+1,\ldots,p} \beta_i E(X_i) + \beta_i X_i, \tag{7.34}$$

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

$$V(E(Y|X_i)) = V(\beta_i X_i) \quad (7.35)$$
$$= \beta_i^2 V(X_i), \quad (7.36)$$

since the variance of a constant term is zero. This immediately leads to the equation 7.33. □

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.

We now discuss various particular values of the first order sensitivity indices of a general function f.

The following proposition allows to support the idea that, if $S_i = 0$, then Y is uncorrelated to X_i *on average*.

Proposition 7.5.3. *Assume that the output Y depends on the input X_i:*

$$Y = f(X_1, X_2, \ldots, X_p) \quad (7.37)$$

where the input variables X_i are independent. If, for some $i = 1, 2, \ldots, p$, the variables Y and X_i are independent, then $S_i = 0$.

Proof. This is an immediate consequence of the definition of S_i. Assume that i is an integer in the set $\{1, 2, \ldots, p\}$ and assume that Y and X_i are independent. Then the random variable $E(Y|X_i)$ does not depend on X_i. Therefore, the conditionnal expectation $E(Y|X_i)$ is a constant. Hence, its variance is zero. □

We emphasize that the previous result is true only *on average*. We also emphasize that the converse is not true, i.e. the equality $S_i = 0$ does not imply that Y and X_i are independent. This is because it might happen that the variable X_i interact with some other variable X_j, with $j \neq i$. In this case, the variable X_i will be influential to the output Y by the mean of the interaction. In the section 7.6, we give an example of such a particular case.

TODO : the case $S_i = 1$

7.6 The product of two variables

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

$$Y = X_1 X_2, \quad (7.38)$$

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

$$E(Y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} X_1 X_2 F(X_1, X_2) dx_1 dx_2, \quad (7.39)$$

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

$$F(x_1, x_2) = F_1(X_1)F_2(X_2), \tag{7.40}$$

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

$$\begin{aligned} E(Y) &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} X_1X_2F_1(X_1)F_2(X_2)dx_1dx_2 && (7.41)\\ &= \left(\int_{-\infty}^{\infty} X_1F_1(X_1)dx_1\right)\left(\int_{-\infty}^{\infty} X_2F_2(X_2)dx_2\right). && (7.42)\\ &= E(X_1)E(X_2). && (7.43) \end{aligned}$$

Therefore,

$$E(Y) = \mu_1\mu_2. \tag{7.44}$$

The variance of Y is

$$V(Y) = E(Y^2) - E(Y)^2. \tag{7.45}$$

The expectation $E(Y^2)$ is

$$\begin{aligned} E(Y^2) &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} (X_1X_2)^2F(x_1, x_2)dx_1dx_2 && (7.46)\\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} (X_1X_2)^2F_1(X_1)F_2(X_2)dx_1dx_2 && (7.47)\\ &= \left(\int_{-\infty}^{\infty} X_1^2F_1(X_1)dx_1\right)\left(\int_{-\infty}^{\infty} X_2^2F_2(X_2)dx_2\right), && (7.48)\\ &= E(X_1^2)E(X_2^2). && (7.49) \end{aligned}$$

Now, we have

$$V(X_1) = E(X_1^2) - E(X_1)^2, \qquad V(X_2) = E(X_2^2) - E(X_2)^2, \tag{7.50}$$

which leads to

$$E(X_1^2) = V(X_1) + E(X_1)^2, \qquad E(X_2^2) = V(X_2) + E(X_2)^2. \tag{7.51}$$

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

$$S_1 = \frac{V(E(Y|X_1))}{V(Y)}, \qquad S_2 = \frac{V(E(Y|X_2))}{V(Y)}. \tag{7.57}$$

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

$$S_1 = \frac{V(\mu_2 X_1)}{V(Y)}, \qquad S_2 = \frac{V(X_2)}{V(Y)}. \tag{7.58}$$

We get

$$S_1 = \frac{\mu_2^2 V(X_1)}{V(Y)}, \qquad S_2 = \frac{\mu_1^2 V(X_2)}{V(Y)}. \tag{7.59}$$

Finally, the first order sensitivity indices are

$$S_1 = \frac{\mu_2^2\sigma_1^2}{V(Y)}, \qquad S_2 = \frac{\mu_1^2\sigma_2^2}{V(Y)}. \tag{7.60}$$

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

$$\mu_2^2\sigma_1^2 + \mu_1^2\sigma_2^2 \leq V(Y) = \sigma_1^2\sigma_2^2 + \sigma_1^2\mu_2^2 + \mu_1^2\sigma_2^2. \tag{7.61}$$

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

$$\frac{\mu_2^2\sigma_1^2}{V(Y)} + \frac{\mu_1^2\sigma_2^2}{V(Y)} \leq 1. \tag{7.62}$$

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

$$S_1 + S_2 \leq 1. \tag{7.63}$$

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 $S_{1,2}$ the sensitivity index associated with the group of variables (X_1, X_2) as

$$S_{1,2} = 1 - S_1 - S_2 = \frac{\sigma_1^2\sigma_2^2}{V(Y)}. \tag{7.64}$$

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.

The following function defines the product of the X_1 and X_2 variables. Each column correspond to an input random variable and each row correspond to a numerical experiment.

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

The following function computes the exact sensitivity indices for the product function.

```
function exact = product_saexact ( mu1 , sigma1 , mu2 , sigma2 )
    // Exact results for the Product function
    exact.expectation = mu1*mu2;
    exact.var = mu2^2*sigma1^2 + mu1^2*sigma2^2 + sigma1^2*sigma2^2;
    // Sensitivity indices.
    exact.S1 = ( mu2^2*sigma1^2 ) / exact.var;
    exact.S2 = ( mu1^2*sigma2^2 ) / exact.var;
    exact.S12 = ( sigma1^2 * sigma2^2 ) / exact.var;
    exact.ST1 = exact.S1 + exact.S12;
    exact.ST2 = exact.S2 + exact.S12;
endfunction
```

The following script computes the exact sensitivity indices in the particular case where 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.

```
// First variable
// Normal
mu1 = 1.5;
sigma1 = 0.5;
// Second variable
// Normal
mu2 = 3.5;
sigma2 = 2.5;
exact = product_saexact ( mu1 , sigma1 , mu2 , sigma2 )
```

The previous script produces the following output.

```
-->exact = product_saexact ( mu1 , sigma1 , mu2 , sigma2 )
 exact  =
   expectation: 5.25
   var: 18.6875
   S1: 0.1638796
   S2: 0.7525084
   S12: 0.0836120
   ST1: 0.2474916
   ST2: 0.8361204
```

We can see that the sum of the two first order indices S_1 and S_2 is lower than 1.

```
-->exact.S1+exact.S2
 ans  =
    0.9163880
```

This is because the interaction between the two variables X_1 and X_2 and is measured by $S_{1,2} = 0.0836120$.

The scatter plots for this function is presented in the figures 7.4 and fig-product-scatter2. These scatter plots are based on a Monte-Carlo simulation with 1000 experiments. It is clear that the variability of Y increases with the magnitude of X_1 or X_2.

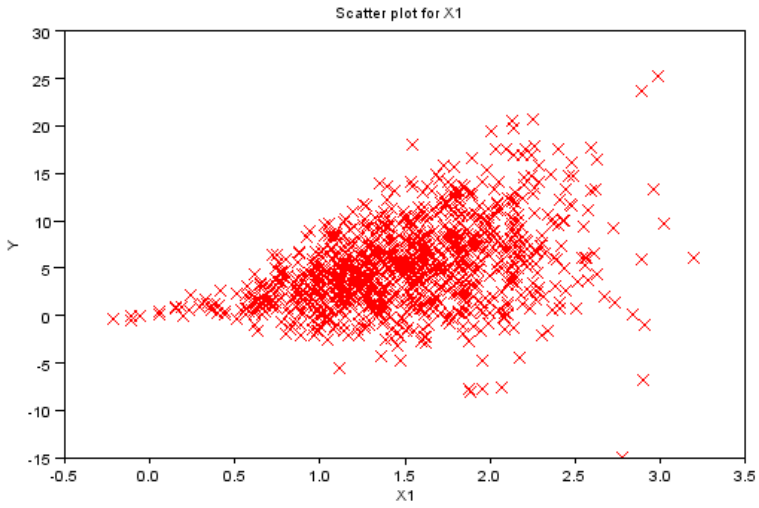

Figure 7.4: Scatter plot for the product function $X_1 \cdot X_2$ - Variable X_1.

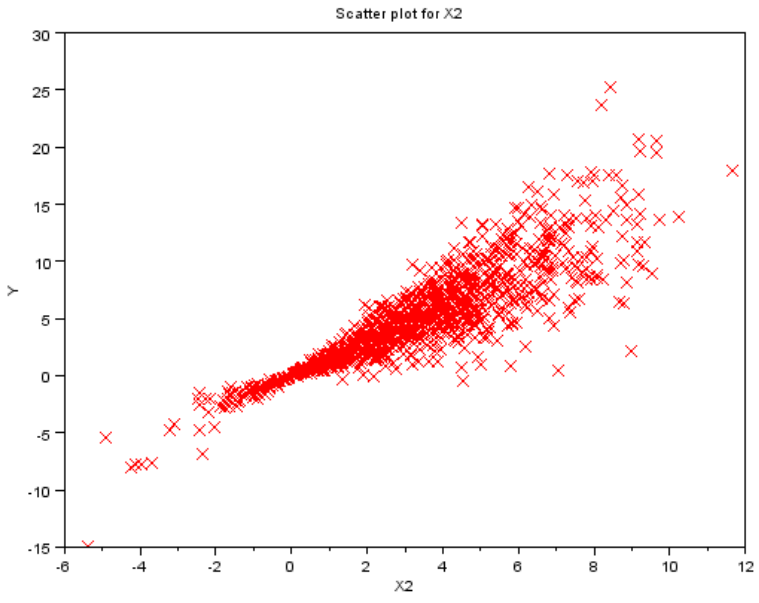

Figure 7.5: Scatter plot for the product function $X_1 \cdot X_2$ - Variable X_2.

The histogram of the output Y is presented in the figure 7.6.

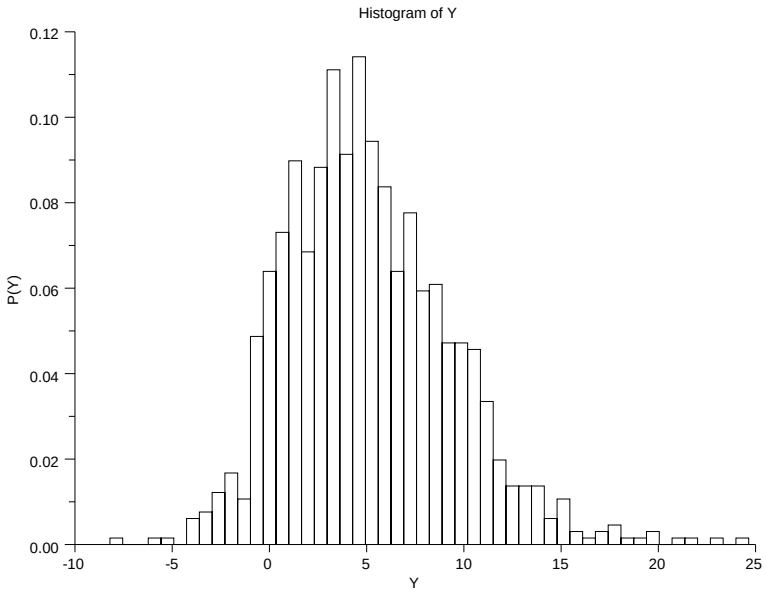

Figure 7.6: Histogram of Y for the product function $X_1 \cdot X_2$.

In the following script, we perform the sensitivity analysis with the NISP library.

```
// 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);
var = polychaos_getvariance(pc);
mprintf("Mean␣␣␣␣=␣%f␣(expectation␣=␣%f)\n",average,exact.expectation);
mprintf("Variance␣␣␣␣=␣%f␣(expectation␣=␣%f)\n",var,exact.var);
mprintf("First␣order␣sensitivity␣index\n");
S1 = polychaos_getindexfirst(pc,1);
mprintf("␣␣␣␣Variable␣X1␣=␣%f␣(expectation␣=␣%f)\n",S1,exact.S1);
re = abs(S1- S1_expectation)/S1_expectation;
mprintf("␣␣␣␣␣␣␣␣Relative␣Error␣=␣%f\n", re);
S2 = polychaos_getindexfirst(pc,2);
mprintf("␣␣␣␣Variable␣X2␣=␣%f␣(expectation␣=␣%f)\n",S2,exact.S2);
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.

We are now considering the fact that $S_i = 0$ does not imply that the output Y is independent of X_i. We consider the case of the product function to show an example of this.

We consider the case where $\mu_2 = 0$. If we plug the equality $\mu_2 = 0$ into the equations 7.60, we get $S_1 = 0$. But the equation $Y = X_1 X_2$ is clear about the fact that Y is not independent of X_1 at all! The following script allows to compute the sensitivity indices in such a case.

```
mu1 = 0.1;
sigma1 = 0.5;
mu2 = 0;
sigma2 = 2.5;
exact = product_saexact ( mu1 , sigma1 , mu2 , sigma2 )
```

The previous script produces the following output.

```
-->exact = product_saexact ( mu1 , sigma1 , mu2 , sigma2 )
 exact  =
   expectation: 0
   var: 1.625
   S1: 0
   S2: 0.0384615
   S12: 0.9615385
   ST1: 0.9615385
   ST2: 1
```

We see that $S_1 = 0$ and S_2 is small. There again, the effect on the output Y is caused by the interaction between X_1 and X_2, which is measured by $S_{1,2} = 0.9615385$. In other words, 96% of the variance of the output Y is caused by the interaction of X_1 and X_2.

The figure 7.7 presents the scatter plot of the output variable Y. The fact that X_2 has a zero mean implies that, whatever the value of X_1, the expectation value of Y is zero, i.e. $E(Y|X_1 = \hat{x_1}) = 0$, for any $\hat{x_1}$. As a result, the variance $V(E(Y|X_1)) = 0$. This can be seen in the figure 7.7. The conditionnal expectation $E(Y|X_1)$ is the mean of Y given a value of X_1 and so can be computed by averaging the values of Y on a vertical line $X_1 = \hat{x_1}$. Since the values of Y are symetrically dispersed under and over the line $Y = 0$, the expectation $E(Y|X_1)$ is zero. Hence the variance of the averaged values of Y is also zero.

7.7 Sobol decomposition

Sobol [10] 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.

Proposition 7.7.1. *(*Sobol decomposition*) Consider the function f*

$$Y = f(x_1, x_2, \ldots, x_p), \tag{7.65}$$

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

$$Y = f_0 + \sum_{i=1,2,\ldots,p} f_i(x_i) + \sum_{1 \leq i < j \leq p} f_{i,j}(x_i, x_j) + \ldots + f_{1,2,\ldots,p}(x_1, x_2, \ldots, x_p), \tag{7.66}$$

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

$$\int_0^1 f_{i_1,\ldots,i_s}(x_{i_1}, \ldots, x_{i_s}) dx_{i_k} = 0, \tag{7.67}$$

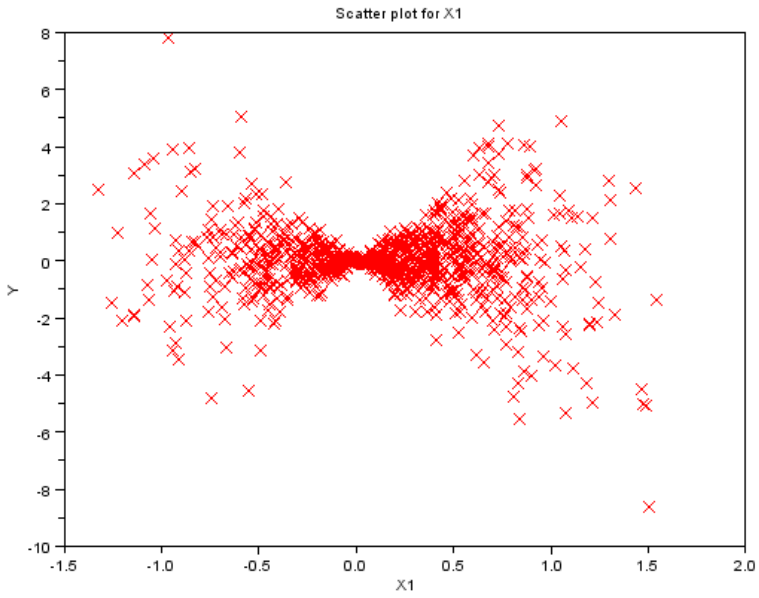

Figure 7.7: Scatter plot for the product function $X_1 \cdot X_2$ when $S_1 = 0$ - Variable X_1.

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

Example *(Sobol decomposition)* Consider the case $p = 3$, i.e. consider the function

$$Y = f(x_1, x_2, x_3), \tag{7.68}$$

where $x_1, x_2, x_3 \in [0, 1]$. The Sobol decomposition theorem states that there exists functions $f_1, f_2, f_3, f_{12}, f_{13}, f_{23}$ and f_{123} such that

$$\begin{aligned} f(x_1, x_2, x_3) &= f_1(x_1) + f_2(x_2) + f_3(x_3) + f_{12}(x_1, x_2) + & (7.69)\\ & f_{13}(x_1, x_3) + f_{23}(x_2, x_3) + f_{123}(x_1, x_2, x_3), & (7.70) \end{aligned}$$

such that the integrals of f_1, f_2 and f_3 satisfy

$$\int_0^1 f_1(x_1)dx_1 = 0, \qquad \int_0^1 f_2(x_2)dx_2 = 0, \qquad \int_0^1 f_3(x_3)dx_3 = 0, \tag{7.71}$$

such that the integrals of f_{12}, f_{13} and f_{23} satisfy

$$\int_0^1 f_{12}(x_1, x_2)dx_1 = 0, \qquad \int_0^1 f_{12}(x_1, x_2)dx_2 = 0, \tag{7.72}$$

$$\int_0^1 f_{13}(x_1, x_3)dx_1 = 0, \qquad \int_0^1 f_{13}(x_1, x_3)dx_3 = 0, \tag{7.73}$$

$$\int_0^1 f_{23}(x_2, x_3)dx_2 = 0, \qquad \int_0^1 f_{23}(x_2, x_3)dx_3 = 0, \tag{7.74}$$

and such that the integrals of f_{123} satisfy

$$\int_0^1 f_{123}(x_1, x_2, x_3)dx_1 = 0, \qquad \int_0^1 f_{123}(x_1, x_2, x_3)dx_2 = 0, \tag{7.75}$$

$$\int_0^1 f_{123}(x_1, x_2, x_3)dx_3 = 0. \tag{7.76}$$

The equalities 7.67 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.

$$\int_0^1 f_{i_1,\dots,i_s}(x_{i_1}, \dots, x_{i_s}) f_{j_1,\dots,j_s}(x_{j_1}, \dots, x_{j_s})dx_1 \dots dx_p = 0, \tag{7.77}$$

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

This because if the two set of indices $(i_1, \dots, i_s)$ and $(j_1, \dots, j_s)$, this means that there is at least one index k which appears in one index and not in the other. By the equality 7.67, this implies that if we integrate with respect to x_k, then the integral is zero. Since the integral in 7.77 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 7.66 with respect to all the variables, we get

$$\int_0^1 f(x)dx = f_0. \tag{7.78}$$

Let us denote by $x_{\sim i}$ the vector x without its i-th component, i.e.

$$x_{\sim i} = (x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots, x_p). \tag{7.79}$$

If we integrate the equation 7.66 with respect to all the variables except i, we get

$$\int_0^1 f(x)dx_{\sim i} = f_0 + f_i(x_i). \tag{7.80}$$

If we integrate the equation 7.66 with respect to all the variables except i and j, we get

$$\int_0^1 f(x)dx_{\sim i,j} = f_0 + f_i(x_i) + f_j(x_j) + f_{i,j}(x_i, x_j). \tag{7.81}$$

If we integrate the equation 7.66 with respect to all the variables except i and j and k, we get

$$\int_0^1 f(x)dx_{\sim i,j,k} = f_0 + f_i(x_i) + f_j(x_j) + f_k(x_k) + \tag{7.82}$$

$$f_{i,j}(x_i, x_j) + f_{i,k}(x_i, x_k) + f_{j,k}(x_j, x_k) + f_{i,j,k}(x_i, x_j, x_k). \tag{7.83}$$

The previous computations allows to get the decomposition functions.

$$
\begin{aligned}
f_0 &= \int_0^1 f(x)dx && (7.84)\\
f_i(x_i) &= -f_0 + \int_0^1 f(x)dx_{\sim i} && (7.85)\\
f_{i,j}(x_i,x_j) &= -f_0 - f_i(x_i) - f_j(x_j) + \int_0^1 f(x)dx_{\sim i,j}, && (7.86)\\
f_{i,j,k}(x_i,x_j,x_k) &= -f_0 - f_i(x_i) - f_j(x_j) - f_k(x_k) - f_{i,j}(x_i,x_j) - f_{i,k}(x_i,x_k) && (7.87)\\
&\quad -f_{j,k}(x_j,x_k) + \int_0^1 f(x)dx_{\sim i,j,k}, && (7.88)
\end{aligned}
$$

until the last term

$$
\begin{aligned}
f_{1,2,\ldots,p}(x_1,x_2,\ldots,x_p) &= f(x) - f_0 - \sum_{i=1,2,\ldots,p} f_i(x_i) - \ldots && (7.89)\\
&\quad - \sum_{1\leq i_1<\ldots<i_{p-1}\leq p} f_{i_1,\ldots,i_{p-1}}(x_{i_1},\ldots,x_{i_{p-1}}). && (7.90)
\end{aligned}
$$

The last term is obviously so that the equality 7.66 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 7.66 is still valid.

7.8 Decomposition of the expectation

We can consider the decomposition 7.66 in terms of expectation and variance.

If we compute the expectation of Y by the expression 7.66 we get

$$E(Y) = f_0, \quad (7.91)$$

which is an obvious consequence of the zero integral property 7.67.

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

$$E(Y|X_i) = \int_0^1 f(x)dx_{\sim i}, \quad (7.92)$$

for all $i = 1,2,\ldots,p$, the equation 7.85 can be written as

$$f_i(x_i) = -f_0 + E(Y|X_i). \quad (7.93)$$

We now plug the equation 7.91 into the previous equality, and get

$$f_i(x_i) = E(Y|X_i) - E(Y). \quad (7.94)$$

Similarly, 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

$$E(Y|X_i, X_j) = \int_0^1 f(x)dx_{\sim i,j}, \tag{7.95}$$

for all $i = 1, 2, \ldots, p$, the equation 7.86 can be written as

$$\begin{aligned} f_{i,j}(x_i, x_j) &= -f_0 - f_i(x_i) - f_j(x_j) + E(Y|X_i, X_j), & (7.96)\\ &= E(Y|X_i, X_j) - E(Y) - E(Y|X_i) - E(Y|X_j). & (7.97) \end{aligned}$$

Similarly, the equation 7.88 leads to

$$\begin{aligned} f_{i,j,k}(x_i, x_j, x_k) &= -f_0 - f_i(x_i) - f_j(x_j) - f_k(x_k) - f_{i,j}(x_i, x_j) - f_{i,k}(x_i, x_k) & (7.98)\\ &\quad -f_{j,k}(x_j, x_k) + E(Y|X_i, X_j, X_k). & (7.99)\\ &= E(Y|X_i, X_j, X_k) - E(Y) - E(Y|X_i) - E(Y|X_j) - E(Y|X_k) & (7.100)\\ &\quad -E(Y|X_i, X_j) - E(Y|X_i, X_k) - E(Y|X_j, X_k). & (7.101) \end{aligned}$$

7.9 Decomposition of the variance

The variance of the function 7.65 can be decomposed into

$$V(Y) = \sum_{i=1}^{p} V_i + \sum_{1\le i<j\le p} V_{ij} + \ldots + V_{1,2,\ldots,p}, \tag{7.102}$$

where

$$\begin{aligned} V_i &= V(E(Y|X_i)), & (7.103)\\ V_{ij} &= V(E(Y|X_i, X_j)) - V_i - V_j & (7.104)\\ V_{i,j,k} &= V(E(Y|X_i, X_j, X_k)) - V_{ij} - V_{ik} - V_{jk} - V_i - V_j - V_k, & (7.105)\\ &\ldots & (7.106)\\ V_{1,2,\ldots,p} &= V(Y) - \sum_{i=1}^{p} V_i - \sum_{1\le i,j\le p} V_{ij} - \sum_{1\le i_1<i_2<\ldots<i_{p-1}\le p} V_{i_1 i_2 \ldots i_p}. & (7.107) \end{aligned}$$

The previous equality is straigtforward, since the equality 7.107 is so that the equality 7.102 must be satisfied, i.e. the term $V_{1,2,\ldots,p}$ is the difference between $V(Y)$ and all the variances associated with lower orders.

It can be proved that the terms of the variance decomposition 7.102 are the variances of the functions defined by 7.91, 7.94, 7.97, 7.101, i.e.

$$V_{i_1,i_2,\ldots,i_s} = V(f_{i_1,i_2,\ldots,i_s}(X_{i_1}, X_{i_2}, \ldots, X_{i_s})), \tag{7.108}$$

for all indices $\{i_1, i_2, \ldots, i_p\} \in \{1, 2, \ldots, p\}$. We can expand the previous equality to make it more explicit.

$$\begin{aligned} V_i &= V(f_i(X_i)), && (7.109)\\ V_{ij} &= V(f_{i,j}(X_i, X_j)), && (7.110)\\ V_{ijk} &= V(f_{i,j,k}(X_i, X_j, k)), && (7.111)\\ \ldots & && (7.112)\\ V_{1,2,\ldots,p} &= V(f_{1,2,\ldots,p}(X_1, X_2, \ldots, X_p)). && (7.113) \end{aligned}$$

7.10 Higher order sensitivity indices

The previous decomposition of the variance can be used in order to compute sensitivity indices beyond the first order indices.

Definition 7.10.1. *(*High order sensitivity indices*) The sensitivity indices are equal to*

$$\begin{aligned} S_i &= = \frac{V_i}{V(Y)}, && (7.114)\\ S_{ij} &= = \frac{V_{ij}}{V(Y)}, && (7.115)\\ S_{ijk} &= = \frac{V_{ijk}}{V(Y)}, && (7.116)\\ \ldots & && (7.117)\\ S_{1,2,\ldots,p} &= = \frac{V_{1,2,\ldots,p}}{V(Y)}. && (7.118) \end{aligned}$$

The order 2 sensitivity index S_{ij} is associated with the sensitivity of the variance of the output Y to the interaction of the inputs X_i and X_j, which is not taken into account by the effect of the variables X_i and X_j alone. The order 3 sensitivity index S_{ijk} is associated with the sensitivity of the variance of the output Y to the interaction of the inputs X_i, X_j and X_k, which is not taken into account neither by the effect of the variables alone, nor by the interactions of two variables.

Consider a function f with three inputs, i.e. $p = 3$. There are three sensitivity indices of order 1, i.e. S_1, S_2 and S_3, three sensitivity indices of order 2, i.e. S_{12}, S_{13} and S_{23}, and one sensitivity index of order 3, i.e. S_{123}.

Proposition 7.10.2. *(*Number of sensitivity indices*) The total number of sensitivity indices is* $2^p - 1$.

Proof. We can count the number of sensitivity indices of a general model with p input random variables. There are obviously p sensitivity indices of order one. This corresponds to computing the list of subsets made of 1 index in the set $\{1, 2, \ldots, p\}$, which implies that that the number of first order indices is $\binom{p}{1} = p$. The number of second order indices corresponds to computing the list of subsets made of 2 indices in the set $\{1, 2, \ldots, p\}$, which implies that that the number of second order indices is $\binom{p}{2}$. Similarily, the number of third order indices is $\binom{p}{3}$. In the end,

the number of p order sensitivity indices is $\binom{p}{p}$. Hence, the total number of sensitivity indices is

$$\binom{p}{1} + \binom{p}{2} + \ldots + \binom{p}{p}. \tag{7.119}$$

On the other hand, we know that

$$\binom{p}{0} + \binom{p}{1} + \binom{p}{2} + \ldots + \binom{p}{p} = 2^p. \tag{7.120}$$

Since $\binom{p}{0} = 1$, we can conclude that the total number of sensitivity indices is $2^p - 1$. □

The equation 7.102 states that the sum of the partial variances V_i, $V_{i,j}$, $V_{i,j,k}$, etc... is equal to the variance $V(Y)$. We can divide the equation 7.102 by $V(Y)$ and get

$$1 = \sum_{i=1}^{p} \frac{V_i}{V(Y)}) + \sum_{1 \leq i < j \leq p} \frac{V_{ij}}{V(Y)}) + \ldots + \frac{V_{1,2,\ldots,p}}{V(Y}). \tag{7.121}$$

Hence, the sum of the sensitivity indices is equal to 1, i.e.

$$1 = \sum_{i=1}^{p} S_i + \sum_{1 \leq i < j \leq p} S_{i,j} + \ldots + S_{1,2,\ldots,p}. \tag{7.122}$$

Example *(High order sensitivity indices)* Consider the case $p = 3$. We have

$$1 \quad = \quad S_1 + S_2 + S_3 + S_{12} + S_{23} + S_{13} + S_{123}. \tag{7.123}$$

The total variance of Y is

$$V(Y) = V(E(Y|X_i)) + E(V(Y|X_i)), \tag{7.124}$$

which implies

$$1 = \frac{V(E(Y|X_i))}{V(Y)} + \frac{E(V(Y|X_i))}{V(Y)}. \tag{7.125}$$

The term $\frac{V(E(Y|X_i))}{V(Y)}$ in the previous equality is S_i, the first order sensitivity indice for X_i. Hence

$$S_i = 1 - \frac{E(V(Y|X_i))}{V(Y)}. \tag{7.126}$$

7.11 Total sensitivity indices

In this section, we present the total sensitivity indices.

We are interested in all the sensitivity indices S_i, S_{ij}, S_{ijk}, etc... associated with a given variable X_i. For example, we are interested in the sensitivity indices associated with X_2, in a function with $p = 3$ parameters. In this case, the associated sensitivity indices are S_2, S_{12}, S_{23} and S_{123}. The associated indices are (2), (12), (23) and (123), that is, all the indices containing 2.

Let us denote by M_i, the set of s-tuples $(i_1, i_2, \ldots, i_s)$ containing i, with $s \leq p$. More formally, M_i is the set of s-tuples $(i_1, i_2, \ldots, i_s)$ such that $i_1, i_2, \ldots, i_s \in \{1, 2, \ldots, p\}$ and there is a $j \in \{1, 2, \ldots, s\}$ such that $i_j = i$. In the previous example, we have $C_2 = \{(2), (12), (23), (123)\}$.

Definition 7.11.1. *(*Total sensitivity indices*) For any $i = 1, 2, \ldots, p$, the total sensitivity index ST_i with respect to the variable X_i is the sum of all the sensitivity indices associated with the variable X_i, i.e.*

$$ST_i = \sum_{k \in M_i} S_k. \tag{7.127}$$

Example *(Total sensitivity indices)* Consider the case $p = 3$. The total sensitivity index associated with X_2 is

$$ST_2 = S_2 + S_{12} + S_{23} + S_{123}. \tag{7.128}$$

By the equation 7.122, we have

$$\begin{aligned} 1 &= S_1 + S_2 + S_3 + S_{12} + S_{23} + S_{13} + S_{123} && (7.129)\\ &= ST_2 + S_1 + S_3 + S_{13}. && (7.130) \end{aligned}$$

In other words, the sum of the total sensitivity index with respect to X_2 and the sensitivity indices not containing 2 is 1. Hence,

$$ST_2 = 1 - S_1 - S_3 - S_{13}. \tag{7.131}$$

We can make a link between the total sensitivity indice ST_i and the expectation of the conditionnal variance of the variables different from i. We have

$$1 = \frac{V(E(Y|X_{\sim i}))}{V(Y)} + \frac{E(V(Y|X_{\sim i}))}{V(Y)}. \tag{7.132}$$

The term $\frac{E(V(Y|X_{\sim i}))}{V(Y)}$ is the total variance ST_i for the variable X_i. Hence,

$$ST_i = 1 - \frac{V(E(Y|X_{\sim i}))}{V(Y)}. \tag{7.133}$$

7.12 Ishigami function

In this section, we consider the model

$$Y = f(X_1, X_2, X_3) = \sin(X_1) + a\sin^2(X_2) + bX_3^4\sin(X_1) \tag{7.134}$$

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

$$f_i(X_i) = \frac{1}{2\pi}, \tag{7.135}$$

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.

7.12.1 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,

$$\int_{-\pi}^{\pi} \sin(x)dx = 0. \tag{7.136}$$

We are going to prove that

$$\int_{-\pi}^{\pi} \sin^2(x)dx = \pi. \tag{7.137}$$

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

$$\begin{aligned} \int_{-\pi}^{\pi} \sin^2(x)dx &= [-\cos(x)\sin(x)]_{-\pi}^{\pi} - \int_{-\pi}^{\pi} (-\cos(x))\cos(x)dx & (7.138)\\ &= 0 + \int_{-\pi}^{\pi} \cos^2(x)dx. & (7.139) \end{aligned}$$

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

$$\begin{aligned} \int_{-\pi}^{\pi} \sin^2(x)dx &= \int_{-\pi}^{\pi} (1 - \cos^2(x))dx & (7.140)\\ &= 2\pi - \int_{-\pi}^{\pi} \cos^2(x)dx. & (7.141) \end{aligned}$$

We now combine 7.139 and 7.141 and get

$$\int_{-\pi}^{\pi} \cos^2(x)dx = 2\pi - \int_{-\pi}^{\pi} \cos^2(x)dx. \tag{7.142}$$

The previous equality implies that

$$2\int_{-\pi}^{\pi} \cos^2(x)dx = 2\pi, \tag{7.143}$$

which leads to

$$\int_{-\pi}^{\pi} \cos^2(x)dx = \pi. \tag{7.144}$$

Finally, the previous equality, combined with 7.139 immediately leads to 7.137.

We are going to prove that

$$\int_{-\pi}^{\pi} \sin^4(x)dx = \frac{3\pi}{4}. \tag{7.145}$$

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

$$\begin{aligned} \int_{-\pi}^{\pi} \sin^4(x)dx &= \left[-\cos(x)\sin^3(x)\right]_{-\pi}^{\pi} - \int_{-\pi}^{\pi}(-\cos(x))(3\sin^2(x)\cos(x))dx & (7.146)\\ &= 0 + 3\int_{-\pi}^{\pi} \cos^2(x)\sin^2(x)dx. & (7.147) \end{aligned}$$

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

$$\begin{aligned} \int_{-\pi}^{\pi} \sin^4(x)dx &= \int_{-\pi}^{\pi} \sin^2(x)\sin^2(x)dx & (7.148)\\ &= \int_{-\pi}^{\pi} (1-\cos^2(x))\sin^2(x)dx & (7.149)\\ &= \int_{-\pi}^{\pi} \sin^2(x)dx - \int_{-\pi}^{\pi} \cos^2(x)\sin^2(x)dx. & (7.150) \end{aligned}$$

We plug the equality 7.137 into the previous equation and get

$$\int_{-\pi}^{\pi} \sin^4(x)dx = \pi - \int_{-\pi}^{\pi} \cos^2(x)\sin^2(x)dx. \tag{7.151}$$

We combine 7.147 and 7.147 and get

$$3\int_{-\pi}^{\pi} \cos^2(x)\sin^2(x)dx = \pi - \int_{-\pi}^{\pi} \cos^2(x)\sin^2(x)dx. \tag{7.152}$$

The previous equation leads to

$$4\int_{-\pi}^{\pi} \cos^2(x)\sin^2(x)dx = \pi, \tag{7.153}$$

which implies

$$\int_{-\pi}^{\pi} \cos^2(x)\sin^2(x)dx = \frac{\pi}{4}, \tag{7.154}$$

We finally plug the equation 7.154 into 7.147 and get the equation 7.145.

7.12.2 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.

$$g_{1,2,3}(x_1, x_2, x_3) = g_1(x_1)g_2(x_2)g_3(x_3). \tag{7.155}$$

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

$$\begin{aligned}
E(\sin(X_1)) &= \int_{-\pi}^{\pi}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} \sin(x_1) g_{1,2,3}(x_1, x_2, x_3) dx_1 dx_2 dx_3 && (7.156)\\
&= \int_{-\pi}^{\pi}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} \sin(x_1) g_1(x_1) g_2(x_2) g_3(x_3) dx_1 dx_2 dx_3 && (7.157)\\
&= \frac{1}{2\pi}\int_{-\pi}^{\pi} \sin(x_1) dx_1 && (7.158)\\
&= 0. && (7.159)
\end{aligned}$$

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

$$\begin{aligned}
E(\sin^2(X_2)) &= \int_{-\pi}^{\pi}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} \sin^2(x_2) g_{1,2,3}(x_1, x_2, x_3) dx_1 dx_2 dx_3 && (7.160)\\
&= \int_{-\pi}^{\pi} \sin^2(x_2) g_2(x_2) dx_2 && (7.161)\\
&= \frac{1}{2\pi}\int_{-\pi}^{\pi} \sin^2(x_2) dx_2. && (7.162)
\end{aligned}$$

The equality 7.137 then implies that

$$\begin{aligned}
E(\sin^2(X_2)) &= \frac{1}{2\pi}\cdot\pi && (7.163)\\
&= \frac{1}{2}. && (7.164)
\end{aligned}$$

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

$$\begin{aligned}
E(X_3^4) &= \int_{-\pi}^{\pi}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} x_3^4 g_{1,2,3}(x_1, x_2, x_3) dx_1 dx_2 dx_3 && (7.165)\\
&= \int_{-\pi}^{\pi}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi} x_3^4 g_1(x_1) g_2(x_2) g_3(x_3) dx_1 dx_2 dx_3 && (7.166)\\
&= \int_{-\pi}^{\pi} x_3^4 g_3(x_3) dx_3 && (7.167)\\
&= \frac{1}{2\pi}\int_{-\pi}^{\pi} x_3^4 dx_3, && (7.168)\\
&= \frac{1}{2\pi}\left[\frac{1}{5}x_3^5\right]_{-\pi}^{\pi} && (7.169)\\
&= \frac{1}{2\pi}\left(\frac{1}{5}\pi^5 - \frac{1}{5}(-\pi)^5\right) && (7.170)\\
&= \frac{1}{2\pi}\frac{2}{5}\pi^5 && (7.171)\\
&= \frac{1}{5}\pi^4. && (7.172)
\end{aligned}$$

We are now going to use the expectations 7.159, 7.164 and 7.172 in order to compute the expectation of the output Y. The model 7.134 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

$$\begin{aligned} E(Y) &= E(\sin(X_1)) + E(a\sin^2(X_1)) + E(bX_3^4\sin(X_1)) && (7.173)\\ &= E(\sin(X_1)) + aE(\sin^2(X_1)) + bE(X_3^4\sin(X_1)). && (7.174)\end{aligned}$$

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

$$\begin{aligned} E(X_3^4\sin(X_1)) &= \int_{-\pi}^{\pi}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi}(x_3^4\sin(x_1))g_{1,2,3}(x_1,x_2,x_3)dx_1dx_2dx_3 && (7.175)\\ &= \int_{-\pi}^{\pi}\int_{-\pi}^{\pi}\int_{-\pi}^{\pi}(x_3^4\sin(x_1))g_1(x_1)g_2(x_2)g_3(x_3)dx_1dx_2dx_3 && (7.176)\\ &= E(X_3^4)E(\sin(X_1)). && (7.177)\end{aligned}$$

Hence, the expectation of Y is

$$E(Y) = E(\sin(X_1)) + aE(\sin^2(X_1)) + bE(X_3^4)E(\sin(X_1)). \quad (7.178)$$

We now combine the equations 7.159, 7.164 and 7.172 and get

$$\begin{aligned} E(Y) &= 0 + a\frac{1}{2} + b\frac{1}{5}\pi^4\cdot 0 && (7.179)\\ &= \frac{a}{2}. && (7.180)\end{aligned}$$

7.12.3 Variance

The variance of the output Y is

$$\begin{aligned} V(Y) &= E(Y^2) - E(Y)^2 && (7.181)\\ &= E\left(\left(\sin(X_1) + a\sin^2(X_2) + bX_3^4\sin(X_1)\right)^2\right) - E(Y)^2 && (7.182)\\ &= E\left(\sin^2(X_1) + a^2\sin^4(X_2) + b^2X_3^8\sin^2(X_1) + \right. && (7.183)\\ &\quad 2\sin(X_1)a\sin^2(X_2) + 2\sin^2(X_1)bX_3^4 + && (7.184)\\ &\quad \left. 2a\sin^2(X_2)bX_3^4\sin(X_1)\right) - E(Y)^2 && (7.185)\\ &= E(\sin^2(X_1)) + a^2E(\sin^4(X_2)) + b^2E(X_3^8)E(\sin^2(X_1)) + && (7.186)\\ &\quad 2aE(\sin(X_1))E(\sin^2(X_2)) + 2bE(\sin^2(X_1))E(X_3^4) + && (7.187)\\ &\quad 2abE(\sin^2(X_2))E(X_3^4)E(\sin(X_1)) - E(Y)^2. && (7.188)\end{aligned}$$

By the equality 7.159, 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

$$\begin{aligned} V(Y) &= E(\sin^2(X_1)) + a^2E(\sin^4(X_2)) + b^2E(X_3^8)E(\sin^2(X_1)) + && (7.189)\\ &\quad 2bE(X_3^4)E(\sin^2(X_1)) - E(Y)^2 && (7.190)\end{aligned}$$

We now compute the terms appearing in the previous equality. Actually, we do not have much to compute, since the equalities 7.164 and 7.172 are already available. Indeed, the equality 7.164 immediately leads to

$$\begin{aligned} E(\sin^2(X_1)) &= E(\sin^2(X_2)) && (7.191)\\ &= \frac{1}{2}. && (7.192) \end{aligned}$$

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

$$\begin{aligned} E(X_3^8) &= \frac{1}{2\pi}\int_{-\pi}^{\pi} x_3^8 dx_3, && (7.193)\\ &= \frac{1}{2\pi}\left[\frac{1}{9}x_3^9\right]_{-\pi}^{\pi} && (7.194)\\ &= \frac{1}{2\pi}\left(\frac{1}{9}\pi^9 - \frac{1}{9}(-\pi)^9\right) && (7.195)\\ &= \frac{1}{2\pi}\frac{2}{9}\pi^9 && (7.196)\\ &= \frac{1}{9}\pi^8. && (7.197) \end{aligned}$$

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

$$E(\sin^2(X_2)) = \frac{1}{2\pi}\int_{-\pi}^{\pi} \sin^4(x_2) dx_2. \quad (7.198)$$

The equality 7.145 then implies that

$$\begin{aligned} E(\sin^4(X_2)) &= \frac{1}{2\pi}\cdot\frac{3\pi}{4} && (7.199)\\ &= \frac{3}{8}. && (7.200) \end{aligned}$$

We now plug the equalities 7.192, 7.172, 7.197 and 7.200 into 7.190, and get

$$\begin{aligned} V(Y) &= \frac{1}{2} + a^2\frac{3}{8} + b^2\frac{1}{9}\pi^8\frac{1}{2} + 2b\frac{1}{5}\pi^4\frac{1}{2} - \frac{a^2}{2^2} && (7.201)\\ &= \frac{1}{2} + \frac{3a^2}{8} + \frac{b^2\pi^8}{18} + \frac{b\pi^4}{5} - \frac{a^2}{4} && (7.202)\\ &= \frac{1}{2} + \frac{a^2}{8} + \frac{b^2\pi^8}{18} + \frac{b\pi^4}{5} && (7.203) \end{aligned}$$

7.12.4 Sobol decomposition

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

By the equation 7.91, we have

$$\begin{aligned} f_0 &= E(Y) && (7.204)\\ &= \frac{a}{2}. && (7.205) \end{aligned}$$

We are first interested in the first order decomposition functions f_1, f_2 and f_3 and their associated variances V_1, V_2 and V_3.

By the equation 7.85, we have

$$\begin{aligned}
f_1(X_1) &= -f_0 + \int_0^1 \int_0^1 f(x) dx_{\sim 1} && (7.206)\\
&= -\frac{a}{2} + \int_0^1 \int_0^1 f(x) dx_2 dx_3 && (7.207)\\
&= -\frac{a}{2} + \int_0^1 \int_0^1 \left(\sin(X_1) + a\sin^2(x_2) + bx_3^4 \sin(X_1)\right) dx_2 dx_3 && (7.208)\\
&= -\frac{a}{2} + \sin(X_1) + aE(\sin^2(X_2)) + bE(x_3^4)\sin(X_1) && (7.209)\\
&= -\frac{a}{2} + \sin(X_1) + a\frac{1}{2} + bE(X_3^4)\sin(X_1) && (7.210)\\
&= -\frac{a}{2} + \sin(X_1) + a\frac{1}{2} + b\frac{\pi^4}{5}\sin(X_1) && (7.211)\\
&= \sin(X_1)\left(1 + b\frac{\pi^4}{5}\right). && (7.212)
\end{aligned}$$

The variance of f_1 is

$$\begin{aligned}
V_1 &= V(f_1(X_1)) && (7.213)\\
&= E(f_1(X_1)^2) - E(f_1(X_1))^2. && (7.214)
\end{aligned}$$

But the equality 7.67 states that the integral of any function f_i with respect to its arguments is zero. Hence $E(f_1(X_1)) = 0$. This implies

$$\begin{aligned}
V_1 &= E(f_1(X_1)^2) && (7.215)\\
&= E\left(\left(\sin(x_1)\left(1 + b\frac{\pi^4}{5}\right)\right)^2\right) && (7.216)\\
&= E(\sin^2(x_1))\left(1 + b\frac{\pi^4}{5}\right) && (7.217)\\
&= \frac{1}{2}\left(1 + b\frac{\pi^4}{5}\right)^2. && (7.218)
\end{aligned}$$

Similarily, we have

$$\begin{aligned}
f_2(X_2) &= -f_0 + \int_0^1 \int_0^1 f(x) dx_{\sim 2} && (7.219)\\
&= -\frac{a}{2} + \int_0^1 \int_0^1 f(x) dx_1 dx_3 && (7.220)\\
&= -\frac{a}{2} + E(\sin(X_1)) + a\sin^2(X_2) + bE(X_3^4)E(\sin(X_1)) && (7.221)\\
&= -\frac{a}{2} + a\sin^2(X_2). && (7.222)
\end{aligned}$$

Hence,

$$\begin{aligned} V_2 &= E(f_2(X_2)^2) && (7.223)\\ &= E((-\frac{a}{2} + a\sin^2(X_2))^2) && (7.224)\\ &= E(\frac{a^2}{4} + a^2\sin^4(X_2) - a^2\sin^2(X_2)) && (7.225)\\ &= \frac{a^2}{4} + a^2E(\sin^4(X_2)) - a^2E(\sin^2(X_2)) && (7.226)\\ &= \frac{a^2}{4} + \frac{3}{8}a^2 - \frac{a^2}{2} && (7.227)\\ &= \frac{a^2}{8}. && (7.228) \end{aligned}$$

Similarily, we have

$$\begin{aligned} f_3(X_3) &= -f_0 + \int_0^1\int_0^1 f(x)dx_{\sim 3} && (7.229)\\ &= -\frac{a}{2} + \int_0^1\int_0^1 f(x)dx_1dx_2 && (7.230)\\ &= -\frac{a}{2} + E(\sin(X_1)) + aE(\sin^2(X_2)) + bX_3E(\sin(X_1)) && (7.231)\\ &= -\frac{a}{2} + \frac{a}{2} && (7.232)\\ &= 0. && (7.233) \end{aligned}$$

Hence,

$$\begin{aligned} V_3 &= E(f_3(X_3)^2) && (7.234)\\ &= 0. && (7.235) \end{aligned}$$

We can immediately conclude that S_3 is zero. The fact that S_3 is zero illustrates the fact that conditional variances may be difficult to analyze. Indeed, the equation $S_3 = 0$ implies that the fraction of the variance that can be explained by the effect of X_3 alone is zero. It does not imply that the variable X_3 has no effect: X_3 as an effect, when we consider its interaction with X_1, because of the $b\sin(X_1)$ term.

We are now interested in the second order decomposition functions $f_{1,2}$, $f_{1,3}$ and $f_{2,3}$.

We have

$$
\begin{aligned}
f_{1,2}(X_1, X_2) &= -f_0 - f_1(X_1) - f_2(X_2) + \int_0^1 f(x) dx_{\sim 1,2} && (7.236)\\
&= -f_0 - f_1(X_1) - f_2(X_2) + \int_0^1 f(x) dx_3 && (7.237)\\
&= -\frac{a}{2} - \sin(X_1)\left(1 + b\frac{\pi^4}{5}\right) + \frac{a}{2} - a\sin^2(X_2) + && (7.238)\\
&\quad \sin(X_1) + a\sin^2(X_2) + bE(X_3^4)\sin(X_1) && (7.239)\\
&= -\sin(X_1)\left(1 + b\frac{\pi^4}{5}\right) - a\sin^2(X_2) + && (7.240)\\
&\quad \sin(X_1) + a\sin^2(X_2) + b\frac{\pi^4}{5}\sin(X_1) && (7.241)\\
&= 0. && (7.242)
\end{aligned}
$$

Hence,

$$
\begin{aligned}
V_{1,2} &= V(f_{1,2}(X_1, X_2)) && (7.243)\\
&= 0. && (7.244)
\end{aligned}
$$

We can immediately conclude that $S_{1,2}$ is zero. This can be predicted from the equation 7.134, since there is no interaction between the variables X_1 and X_2.

We have

$$
\begin{aligned}
f_{1,3}(X_1, X_3) &= -f_0 - f_1(X_1) - f_3(X_3) + \int_0^1 f(x) dx_{\sim 1,3} && (7.245)\\
&= -f_0 - f_1(X_1) - f_3(X_3) + \int_0^1 f(x) dx_2 && (7.246)\\
&= -\frac{a}{2} - \sin(X_1)\left(1 + b\frac{\pi^4}{5}\right) + && (7.247)\\
&\quad \sin(X_1) + aE(\sin^2(X_2)) + bX_3^4\sin(X_1) && (7.248)\\
&= -\frac{a}{2} - \sin(X_1)\left(1 + b\frac{\pi^4}{5}\right) + \sin(X_1) + \frac{a}{2} + bX_3^4\sin(X_1) && (7.249)\\
&= -\sin(X_1)b\frac{\pi^4}{5} + bX_3^4\sin(X_1) && (7.250)\\
&= b\sin(X_1)\left(X_3^4 - \frac{\pi^4}{5}\right). && (7.251)
\end{aligned}
$$

Hence,

$$
\begin{aligned}
V_{1,3} &= V(f_{1,3}(X_1, X_3)) && (7.252)\\
&= E(f_{1,3}(X_1, X_3)^2) - E(f_{1,3}(X_1, X_3))^2 && (7.253)
\end{aligned}
$$

The equality 7.67 implies that $E(f_{1,3}(X_1, X_3)) = 0$. Hence,

$$\begin{aligned}
V_{1,3} &= E(f_{1,3}(X_1, X_3)^2) && (7.254)\\
&= b^2 E(\sin^2(X_1)) E\left(\left(X_3^4 - \frac{\pi^4}{5}\right)^2\right) && (7.255)\\
&= \frac{b^2}{2} E(X_3^8 + \frac{\pi^8}{25} - 2X_3^4 \frac{\pi^4}{5}) && (7.256)\\
&= \frac{b^2}{2}(E(X_3^8) + \frac{\pi^8}{25} - 2E(X_3^4)\frac{\pi^4}{5}) && (7.257)\\
&= \frac{b^2}{2}(\frac{\pi^8}{9} + \frac{\pi^8}{25} - 2\frac{\pi^4}{5}\frac{\pi^4}{5}) && (7.258)\\
&= \frac{b^2}{2}(\frac{\pi^8}{9} + \frac{\pi^8}{25} - 2\frac{\pi^8}{25}) && (7.259)\\
&= \frac{b^2}{2}(\frac{\pi^8}{9} - \frac{\pi^8}{25}) && (7.260)\\
&= \frac{b^2\pi^8}{2}(\frac{1}{9} - \frac{1}{25}). && (7.261)
\end{aligned}$$

We have

$$\begin{aligned}
f_{2,3}(X_2, X_3) &= -f_0 - f_2(X_2) - f_3(X_3) + \int_0^1 f(x) dx_{\sim 2,3} && (7.262)\\
&= -f_0 - f_2(X_2) - f_3(X_3) + \int_0^1 f(x) dx_1 && (7.263)\\
&= -\frac{a}{2} - \left(-\frac{a}{2} + a\sin^2(X_2)\right) - 0 + && (7.264)\\
&\quad E(\sin(X_1)) + a\sin^2(x_2) + bx_3^4 E(\sin(X_1)) && (7.265)\\
&= 0. && (7.266)
\end{aligned}$$

Hence,

$$V_{2,3} = 0. \quad (7.267)$$

Finally, we can compute the function $f_{1,2,3}$ and the variance $V_{1,2,3}$. We have

$$\begin{aligned}
f_{1,2,3}(X_1, X_2, X_3) &= -f_0 - f_1(X_1) - f_2(X_2) - f_3(X_3) - f_{1,2}(X_1, X_2) && (7.268)\\
&\quad -f_{1,3}(X_1, X_3) - f_{2,3}(X_2, X_3) + f(X_1, X_2, X_3) && (7.269)\\
&= 0. && (7.270)
\end{aligned}$$

Hence

$$V_{1,2,3} = 0. \quad (7.271)$$

7.12.5 Summary of the results

In this section, we present a summary of the results for the Ishigami function

$$Y = f(X_1, X_2, X_3) = \sin(X_1) + a\sin^2(X_2) + bX_3^4 \sin(X_1) \quad (7.272)$$

where X_1, X_2, X_3 are three random variables uniform in $[-\pi, \pi]$. The expectation and the variance of Y are

$$E(Y) = \frac{a}{2} \tag{7.273}$$

$$V(Y) = \frac{1}{2} + \frac{a^2}{8} + \frac{b^2\pi^8}{18} + \frac{b\pi^4}{5} \tag{7.274}$$

The Sobol decomposition functions are

$$f_0 = \frac{a}{2} \tag{7.275}$$

$$f_1(X_1) = \sin(X_1)\left(1 + b\frac{\pi^4}{5}\right) \tag{7.276}$$

$$f_2(X_2) = -\frac{a}{2} + a\sin^2(X_2) \tag{7.277}$$

$$f_3(X_3) = 0 \tag{7.278}$$

$$f_{1,2}(X_1, X_2) = 0 \tag{7.279}$$

$$f_{1,3}(X_1, X_3) = b\sin(X_1)\left(X_3^4 - \frac{\pi^4}{5}\right) \tag{7.280}$$

$$f_{2,3}(X_2, X_3) = 0 \tag{7.281}$$

$$f_{1,2,3}(X_1, X_2, X_3) = 0. \tag{7.282}$$

The Sobol decomposition variances are

$$V_1 = \frac{1}{2}\left(1 + b\frac{\pi^4}{5}\right)^2 \tag{7.283}$$

$$V_2 = \frac{a^2}{8} \tag{7.284}$$

$$V_3 = 0 \tag{7.285}$$

$$V_{1,2} = 0 \tag{7.286}$$

$$V_{1,3} = \frac{b^2\pi^8}{2}(\frac{1}{9} - \frac{1}{25}) \tag{7.287}$$

$$V_{2,3} = 0 \tag{7.288}$$

$$V_{1,2,3} = 0. \tag{7.289}$$

7.12.6 Numerical results

In this section, we present numerical results associated with the Ishigami function.

We begin by defining the Ishigami function.

```
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
```

In the following script, we perform a Monte-Carlo sampling and compute the output of the Ishigami function. Then we plot the output Y against its input arguments X_1, X_2 and X_3.

```
np = 1000;
A = grand(np,3,"def")*2*%pi - %pi;
y = ishigami (A);
scf();
subplot(1,3,1);
plot(A(:,1),y,"bo")
xtitle("","X1","f(X1,X2,X3)");
subplot(1,3,2);
plot(A(:,2),y,"bo")
xtitle("","X2","f(X1,X2,X3)");
subplot(1,3,3);
plot(A(:,3),y,"bo")
xtitle("","X3","f(X1,X2,X3)");
// Put the marks as transparent
h.children(1).children.children.mark_background = 0;
h.children(2).children.children.mark_background = 0;
h.children(3).children.children.mark_background = 0;
```

The previous script produces the figure 7.8.

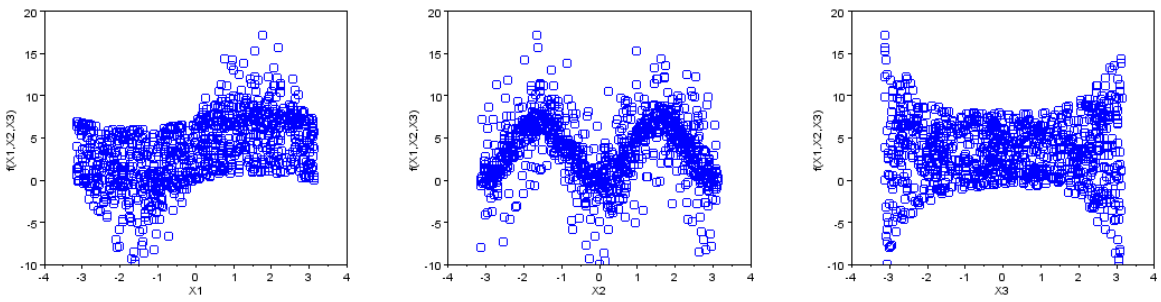

Figure 7.8: Output of the Ishigami function with respect to each input argument.

The following function returns the exact expectation, variance and sensitivity indices of the Ighigami function.

```
function exact = ishigami_saexact ( a , b )
    // Exact results for the Ishigami function
    exact.expectation = a/2;
    exact.var = 1/2 + a^2/8 + b*%pi^4/5 + b^2*%pi^8/18;
    // Sensitivity indices.
    exact.S1 = (1/2 + b*%pi^4/5+b^2*%pi^8/50)/exact.var;
    exact.S2 = (a^2/8)/exact.var;
    exact.S3 = 0;
    exact.S12 = 0;
    exact.S23 = 0;
    exact.S13 = b^2*%pi^8/2*(1/9-1/25)/exact.var;
    exact.S123 = 0;
```

```
    exact.ST1 = exact.S1 + exact.S12 + exact.S13 + exact.S123;
    exact.ST2 = exact.S2 + exact.S12 + exact.S23 + exact.S123;
    exact.ST3 = exact.S3 + exact.S13 + exact.S23 + exact.S123;
endfunction
```

The following script allows to get the results associated with the parameters $a = 7$ and $b = 0.1$.

```
a=7.;
b=0.1;
exact = ishigami_saexact ( a , b )
```

The previous script produces the following output.

```
-->exact = ishigami_saexact ( a , b )
 exact  =
   expectation: 3.5
   var: 13.844588
   S1: 0.3139052
   S2: 0.4424111
   S3: 0
   S12: 0
   S23: 0
   S13: 0.2436837
   S123: 0
   ST1: 0.5575889
   ST2: 0.4424111
   ST3: 0.2436837
```

7.13 A straightforward approach

In this section, we present a straightforward approach for the computation of the first order sensitivity indices. As we are going to see, this method requires a too large number of function evaluations. This section then motivate the need for more elaborate algorithms which are going to be presented in the remaining sections.

Let us compute the sensitivity indice associated with a uniform random variable X_i, for $i = 1, 2, \ldots, p$. The analysis is based on the decomposition of the input variables of the function f

$$f(X) = f(X_i, X_{\sim i}), \tag{7.290}$$

where the indices $\sim i$ are all the integers different from i. By the equation 7.103, we have

$$V_i = V(E(Y|X_i)). \tag{7.291}$$

We could employ a straightforward method to estimate the previous variance, based on two nested loops. Indeed, we can set the variable X_i to a particular value, say $\hat{X}_j$. From there, we can create a sampling for the remaining parameters $X_{\sim i}$, based on n Monte-Carlo simulations, where n is a relatively large integer, say $n = 1000$. This leads to the experiments

$$y_{k|X_i=\hat{X}_j} = f(x_1^k, x_2^k, \ldots, \hat{X}_i, \ldots, x_p^k), \tag{7.292}$$

for $k = 1, 2, \ldots, n$. Once done, we have a set of values $\{y_k | X_i = \hat{X}_k\}_{k=1,2,\ldots,n}$ and can compute the associated expectation

$$\tilde{E}(Y|X_i = \hat{X}_j) = \frac{1}{n} \sum_{k=1,2,\ldots,n} y_{k|X_i=\hat{X}_j}. \tag{7.293}$$

We can repeat this experiment m times, so that we now have the set of expectations $\tilde{E}(Y|X_i = \hat{X}_j)$ for $j = 1, 2, \ldots, m$. We can then compute the expectation and the variance of this random variable, with

$$\tilde{E}(Y) = \frac{1}{m} \sum_{j=1,2,\ldots,n} \tilde{E}(Y|X_i = \hat{X}_j) \tag{7.294}$$

$$\tilde{V}_i = \frac{1}{m-1} \sum_{j=1,2,\ldots,m} (\tilde{E}(Y|X_i = \hat{X}_j) - \tilde{E}(Y))^2. \tag{7.295}$$

This method requires a total of $n \cdot m$ evaluations of f, which is too large to be manageable in practice. For example, if n =1 000 and m =1 000, the previous method requires 1 000 000 function evaluations. Given the accuracy of Monte-Carlo experiments, the result would be unusable in practical cases.

7.14 The Sobol method for sensitivity analysis

In this section, we present a method by Sobol [10] to estimate the sensitivity indices, using Monte-Carlo experiments.

Let us compute the sensitivity indice associated with a uniform random variable X_i, for $i = 1, 2, \ldots, p$. The analysis is based on the decomposition of the input variables of the function f

$$f(X) = f(X_i, X_{\sim i}), \tag{7.296}$$

where the indices $\sim i$ are all the integers different from i. By the equation 7.103, we have

$$V_i = V(E(Y|X_i)). \tag{7.297}$$

By the definition of a variance,

$$V_i = E(E(Y|X_i)^2) - E(E(Y|X_i))^2. \tag{7.298}$$

We can begin by computing the expression $E(E(Y|X_i))^2$. By the definition of the conditionnal expectation,

$$E(Y|X_i) = \int f(X_i, X_{\sim i}) dX_{\sim i}, \tag{7.299}$$

so that

$$E(E(Y|X_i)) = \int \left(\int f(X_i, X_{\sim i}) dX_{\sim i} \right) dX_i \tag{7.300}$$

$$= \int f(X) dX \tag{7.301}$$

$$= E(Y). \tag{7.302}$$

Hence,

$$V_i = E(E(Y|X_i)^2) - E(Y)^2. \tag{7.303}$$

On the other hand,

$$E(E(Y|X_i)^2) = E\left(\left(\int f(X_i, X_{\sim i})dX_{\sim i}\right)^2\right) \tag{7.304}$$

$$= \int \left(\int f(X_i, X_{\sim i})dX_{\sim i}\right)^2 dX_i \tag{7.305}$$

$$= \int\int\int f(X_i, X_{\sim i})f(X_i, Z_{\sim i})dX_{\sim i}dZ_{\sim i}dX_i. \tag{7.306}$$

We plug the previous equation into 7.298 and get

$$V_i = \int\int\int f(X_i, X_{\sim i})f(X_i, Z_{\sim i})dX_{\sim i}dZ_{\sim i}dX_i. - E(Y)^2. \tag{7.307}$$

Remark 7.14.1. *Although the two previous computations seem monstruous at first, they are simply derived from the basic following result. Assume that g is a smooth integrable function of one variable $x \in [0, 1]$. Then, we have the following elementary result:*

$$\left(\int g(x)dx\right)^2 = \left(\int g(x)dx\right)\left(\int g(x)dx\right) \tag{7.308}$$

$$= \left(\int g(x)dx\right)\left(\int g(y)dy\right) \tag{7.309}$$

$$= \left(\int \left(\int g(x)dx\right) g(y)dy\right) \tag{7.310}$$

$$= \int\int g(x)g(y)dxdy. \tag{7.311}$$

Hence, the move from the equation 7.305 to 7.306 is just the application of the previous result applied to the function $g(X_{\sim i}) = f(X_i, Z_{\sim i})$, i.e. considering the variables $X_{\sim i}$.

In [8], Saltelli notices that the equation 7.306 is the expected value of the function F of $2p-1$ variables:

$$F(X_i, X_{\sim i}, Z_{\sim i}) = f(X_i, X_{\sim i})f(X_i, Z_{\sim i}). \tag{7.312}$$

Hence, the integral 7.306 can be evaluated using a single Monte-Carlo loop.

The following method is due to Sobol, Homma and Saltelli. It is based on the use of two sampling A and B, made of n experiments:

$$A = \begin{pmatrix} x_1^{A,1} & x_2^{A,1} & \dots & x_p^{A,1} \\ x_1^{A,2} & x_2^{A,2} & \dots & x_p^{A,2} \\ \vdots & \vdots & & \vdots \\ x_1^{A,n} & x_2^{A,n} & \dots & x_p^{A,n} \end{pmatrix}, \qquad B = \begin{pmatrix} x_1^{B,1} & x_2^{B,1} & \dots & x_p^{B,1} \\ x_1^{B,2} & x_2^{B,2} & \dots & x_p^{B,2} \\ \vdots & \vdots & & \vdots \\ x_1^{B,n} & x_2^{B,n} & \dots & x_p^{B,n} \end{pmatrix}. \tag{7.313}$$

The sampling can, for example, a Monte-Carlo sampling.

The first step of the algorithm is to perform the experiments given in the matrix A to estimate the expectation and variances of Y. We perform the experiments and store the result in the column vector y^A:

$$y_k^A = f(x_1^{A,k}, x_2^{A,k}, \ldots, x_p^{A,k}), \tag{7.314}$$

for $k = 1, 2, \ldots, n$. We can then estimate the expectation of Y by

$$\tilde{E}(Y) = \frac{1}{n} \sum_{k=1,2,\ldots,n} y_k^A, \tag{7.315}$$

and the variance of Y by

$$\tilde{V}(Y) = \frac{1}{n-1} \sum_{k=1,2,\ldots,n} (y_k^A - \tilde{E}(Y))^2. \tag{7.316}$$

In order to estimate the first order sensitivity index S_1, we create a sampling C_1 by using the first column of A and the remaining columns of B:

$$C_1 = \begin{pmatrix} x_1^{A,1} & x_2^{B,1} & \ldots & x_p^{B,1} \\ x_1^{A,2} & x_2^{B,2} & \ldots & x_p^{B,2} \\ \vdots & \vdots & & \vdots \\ x_1^{A,n} & x_2^{B,n} & \ldots & x_p^{B,n} \end{pmatrix}. \tag{7.317}$$

We now perform the experiments given in C_1:

$$y_k^{C_1} = f(x_1^{C_1,k}, x_2^{C_1,k}, \ldots, x_p^{C_1,k}), \tag{7.318}$$

for $k = 1, 2, \ldots, n$. The conditionnal expectation $E(E(Y|X_1)^2)$ can then be estimated by

$$\tilde{U}_1 = \frac{1}{n-1} \sum_{k=1,2,\ldots,n} y_k^A y_k^{C_1}. \tag{7.319}$$

Finally, the sensitivity with respect to X_1 is

$$S_1 = \frac{\tilde{U}_1 - \tilde{E}(Y)}{\tilde{V}(Y)}. \tag{7.320}$$

More generally, we can estimate the first order sensitivity index S_i, for $i = 1, 2, \ldots, p$. We create a sampling C_i by using the i-th column of A and the remaining columns of B:

$$C_i = \begin{pmatrix} x_1^{B,1} & x_2^{B,1} & \ldots & x_i^{A,1} & \ldots & x_p^{B,1} \\ x_1^{B,2} & x_2^{B,2} & \ldots & x_i^{A,2} & \ldots & x_p^{B,2} \\ \vdots & \vdots & & \vdots & & \vdots \\ x_1^{B,n} & x_2^{B,n} & \ldots & x_i^{A,n} & \ldots & x_p^{B,n} \end{pmatrix}. \tag{7.321}$$

We perform the experiments given in C_i:

$$y_k^{C_i} = f(x_1^{C_i,k}, x_2^{C_i,k}, \ldots, x_p^{C_i,k}), \tag{7.322}$$

for $k = 1, 2, \ldots, n$. The conditionnal expectation $E(E(Y|X_i)^2)$ can then be estimated by

$$\tilde{U}_i = \frac{1}{n-1} \sum_{k=1,2,\ldots,n} y_k^A y_k^{C_i}. \tag{7.323}$$

Finally, the sensitivity with respect to X_i is

$$S_i = \frac{\tilde{U}_i - \tilde{E}(Y)^2}{\tilde{V}(Y)}. \tag{7.324}$$

The initial A sampling requires n function evaluations, which allows to compute the expectation and the variance. From there, each sensitivity indice S_i requires n function evaluations. All the first order sensitivity indices can therefore be approximated with $n + pn$ function evaluations.

The same procedure can be used to estimate the higher sensitivity indices. For example, in order to compute the second order sensitivity indice $S_{i,j}$, we use the sampling

$$C_{i,j} = \begin{pmatrix} x_1^{B,1} & x_2^{B,1} & \ldots & x_i^{A,1} & \ldots & x_j^{A,1} & \ldots & x_p^{B,1} \\ x_1^{B,2} & x_2^{B,2} & \ldots & x_i^{A,2} & \ldots & x_j^{A,1} & \ldots & x_p^{B,2} \\ \vdots & \vdots & & \vdots & & \vdots & & \vdots \\ x_1^{B,n} & x_2^{B,n} & \ldots & x_i^{A,n} & \ldots & x_j^{A,1} & \ldots & x_p^{B,n} \end{pmatrix}. \tag{7.325}$$

We perform the experiments given in C_i:

$$y_k^{C_{i,j}} = f(x_1^{C_{i,j},k}, x_2^{C_{i,j},k}, \ldots, x_p^{C_{i,j},k}), \tag{7.326}$$

for $k = 1, 2, \ldots, n$. The conditionnal expectation $E(E(Y|X_i, X_j)^2)$ can then be estimated by

$$\tilde{U}_{i,j} = \frac{1}{n-1} \sum_{k=1,2,\ldots,n} y_k^A y_k^{C_{i,j}}. \tag{7.327}$$

Finally, the sensitivity with respect to X_i and X_j is

$$S_{i,j} = \frac{\tilde{U}_{i,j} - \tilde{E}(Y)}{\tilde{V}(Y)} - S_i - S_j. \tag{7.328}$$

The procedure can be used to estimate the total sensitivity indices ST_i, for $i = 1, 2, \ldots, p$. We learnt in the section 7.11 that

$$ST_i = 1 - \frac{V(E(Y|X_{\sim i}))}{V(Y)}. \tag{7.329}$$

We then have to approximate

$$\begin{aligned} V_{\sim i} &= V(E(Y|X_{\sim i})) && (7.330) \\ &= E(E(Y|X_{\sim i})^2) - E(E(Y|X_{\sim i}))^2 && (7.331) \\ &= E(E(Y|X_{\sim i})^2) - E(Y)^2. && (7.332) \end{aligned}$$

Therefore, we have to compute

$$U_{\sim i} = E(E(Y|X_{\sim i})^2). \tag{7.333}$$

We perform the experiments from the B sampling and store the result in the column vector y^B:

$$y_k^B = f(x_1^{B,k}, x_2^{B,k}, \ldots, x_p^{B,k}), \tag{7.334}$$

for $k = 1, 2, \ldots, n$. Notice that the difference between the matrices B and C_i, only the column i changes. We finally approximate $U_{\sim i}$ by

$$U_{\sim i} = \frac{1}{n-1} \sum_{k=1,2,\ldots,n} y_k^B y_k^{C_i}. \tag{7.335}$$

The equation 7.323 for the computation of $\tilde{U}_i$ is discussed in [9], Chapter 5. "Methods based on decomposing the variance of the output", section 5.9. We can think of A as the "sample" matrix and of B as the "resample" matrix. The term $\tilde{U}_i$ is obtained from the product of values of f computed from the sample matrix times values of f computed from C_i, i.e. a matrix where all factors except X_i are re-sampled.

- If X_i is an influential factor, then high values of y_k^A will be preferentially associated with high values of $y_k^{C_i}$.
- If X_i is the only influential factor (all the others being dummies), then the two values of f will be identical.
- If X_i is a totally non-influential factor, then high and low values of y_k^A will be randomly associated with high and low values of $y_k^{C_i}$.

Therefore, the estimate $\tilde{U}_i$ of the sensitivity of X_i must be larger for an influential factor X_i than for a non-influential one.

TODO : the accuracy issue of the variance

TODO : why using the covariance instead of the difference formula for u_i

TODO : the comment of Saltelli of another formula for u_i in the case of a non-influential factor

7.15 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 function allows to compute the covariance matrix of its two input arguments `x` and `y`.

```
function C = nisp_cov ( x , 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
```

The following function returns the sensitivity index associated with the experiments ya and yc.

```
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
```

The following script allows to perform the analysis.

```
// 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, exact.S1);
// 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, exact.S2);
// 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, exact.S3);
// 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)-s1-s3;
mprintf("S13 : %f (expected = %f)\n", s13, exact.S13);
// Compute the sensitivity index for {X2,X3}
C = B;
C(:,[2 3])=A(:,[2 3]);
yc = ishigami(C);
s23 = sensitivityindex(ya,yc)-s2-s3;
mprintf("S23 : %f (expected = %f)\n", s23, exact.S23);
// Compute the sensitivity index for {X1,X2,X3}
C = B;
C(:,[1 2 3])=A(:,[1 2 3]);
yc = ishigami(C);
s123 = sensitivityindex(ya,yc)-s1-s2-s3-s12-s23-s13;
mprintf("S123 : %f (expected = %f)\n", s123, exact.S123);
// Compute the total sensitivity index for X1
C = A;
C(:,1)=B(:,1);
yc = ishigami(C);
st1 = 1-sensitivityindex(ya,yc);
mprintf("ST1 : %f (expected = %f)\n", st1, exact.ST1);
// Compute the total sensitivity index for X2
C = A;
C(:,2)=B(:,2);
yc = ishigami(C);
st2 = 1-sensitivityindex(ya,yc);
mprintf("ST2 : %f (expected = %f)\n", st2, exact.ST2);
// Compute the total sensitivity index for X3
C = A;
C(:,3)=B(:,3);
yc = ishigami(C);
st3 = 1-sensitivityindex(ya,yc);
mprintf("ST3 : %f (expected = %f)\n", st3, exact.ST3);
//
// Clean-up
randvar_destroy(rvu1);
randvar_destroy(rvu2);
randvar_destroy(rvu3);
setrandvar_destroy(srvu);
```

The previous script produces the following output.

```
S1 : 0.311242 (expected = 0.313905)
S2 : 0.449588 (expected = 0.442411)
S3 : 0.000827 (expected = 0.000000)
S12 : -0.006056 (expected = 0.000000)
S13 : 0.245094 (expected = 0.243684)
S23 : -0.008243 (expected = 0.000000)
S123 : 0.007548 (expected = 0.000000)
```

```
ST1  :  0.557828  (expected  =  0.557589)
ST2  :  0.442837  (expected  =  0.442411)
ST3  :  0.245226  (expected  =  0.243684)
```

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 7.9.

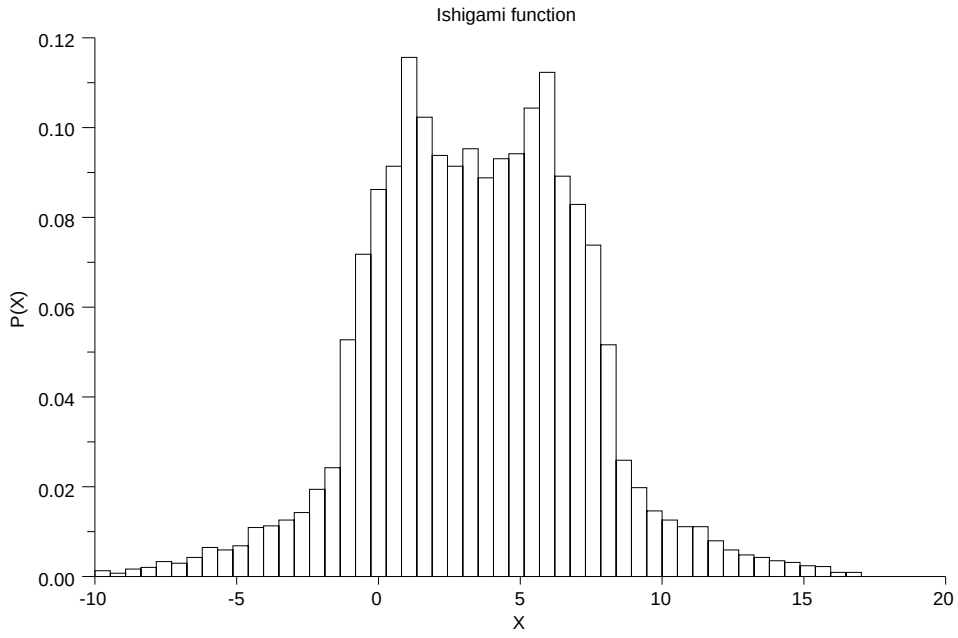

Figure 7.9: Histogram of the output of the Ishigami function.

7.16 Notes and references

More details on this topic can be found in the papers of Homma and Saltelli [4] or in the work of Sobol [10].

The thesis by Jacques [6] presents an overview of sensitivity analysis. Some of the results presented here are extracted from the short introduction by Jacques [5].

The book by Saltelli, Tanrantola, Compolongo and Ratto [9] presnts a description of a few selected techniques for sensitivity analysis. Their presentation is based on the Simlab software, a free development framework for sensitivity analysis and uncertainty analysis. The licence of Simlab is free for non-commercial purposes.

The first figure of this chapter is taken from the slides by Antoniadis [1].

Chapter 8

Thanks

Many thanks to Allan Cornet.

Bibliography

[1] Anestis Antoniadis. Analyse de l'incertitude et analyse de sensibilité - l'analyse de sensibilité, June 2005. Ecole d'été CEA-EDF-INRIA.

[2] Michael Baudin. Programming in scilab, 2010.

[3] Michael Baudin and Jean-Marc Martinez. Polynômes de chaos sous scilab via la librairie nisp. In *42èmes Journées de Statistique, 24 au 28 mai 2010 - Marseille, France*, 2010.

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

[5] Julien Jacques. Analyse de sensibilité globale, 2005. http://math.univ-lille1.fr/~jacques/.

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

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

[8] Andrea Saltelli. Making best use of model evaluations to compute sensitivity indices. *Computer Physics Communications*, 145(2):280 – 297, 2002.

[9] Andrea Saltelli, Stefano Tarantola, Francesca Compolongo, and Marco Ratto. *Sensitivity analysis in practice*. John Wiley and Sons, 2004.

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