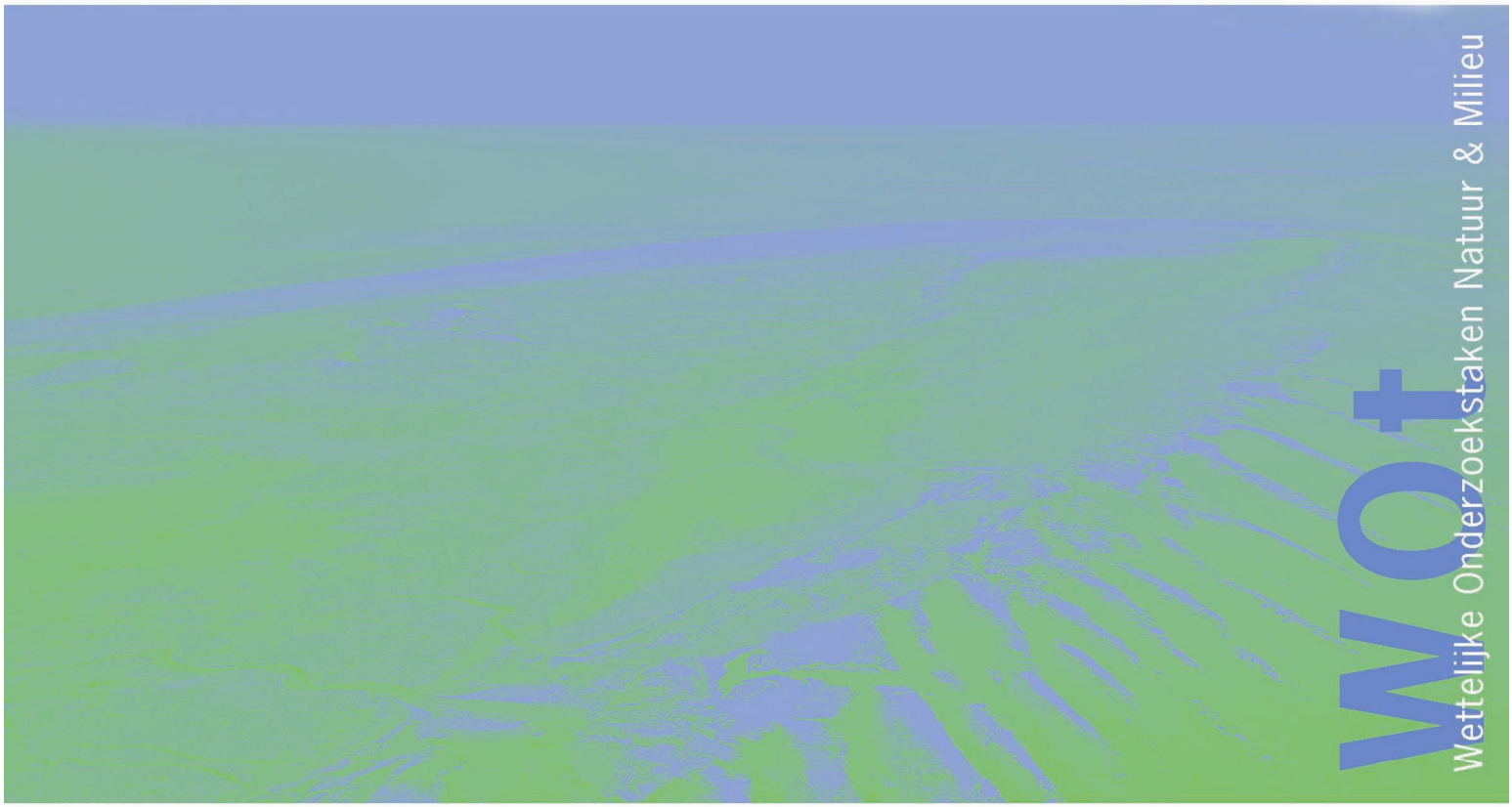
4

ASSA: Algorithms for Stochastic Sensitivity Analysis

Manual for version 1.0

M.J.W. Jansen

werkdocumenten



WOT Wettelijke Onderzoekstaken Natuur & Milieu

WAGENINGEN UR
For quality of life

ASSA: Algorithms for Stochastic Sensitivity Analysis

Manual for version 1.0

M.J.W. Jansen

Werkdocument 4

Wettelijke Onderzoekstaken Natuur & Milieu

Wageningen, november 2005

The 'Working Documents' series presents interim results of research commissioned by the Statutory Research Tasks Unit for Nature & the Environment (WOT N&M) from various external agencies. The series is intended as an internal channel of communication and is not being distributed outside the WOT Unit. The content of this document is mainly intended as a reference for other researchers engaged in projects commissioned by the Unit. As soon as final research results become available, these are published through other channels. The present series includes documents reporting research findings as well as documents relating to research management issues.

Working document 4 has been accepted by Harm Houweling, who commissioned the project on behalf of the Statutory Research Tasks Unit for Nature & the Environment.

Keywords and phrases
Model input uncertainty; model output uncertainty; Monte Carlo; uncertainty analysis; analysis of variance; regression-based sensitivity analysis; regression-free sensitivity analysis; numerical recipes.

Postbus 100, 6700 AC Wageningen.
Tel: +31 (0)317 48 40 85; fax: +31 (0)317 48 35 54; e-mail: Biometris@wur.nl

---

ASSA's manual and source code can be downloaded free of charge from the website of WOT N&M, the Dutch Statutory Research Tasks Unit for Nature & the Environment at Wageningen University and Research Centre. A link to WOT N&M's website will be offered to the SAMO site, which contains references to several software products for sensitivity analysis: http://sensitivity-analysis.jrc.cec.eu.int/forum/default.asp.

The Working Documents series is published by the Statutory Research Tasks Unit for Nature & the Environment (WOT N&M), part of Wageningen UR. This report is available from the secretary's office, and can be downloaded from www.wotnatuurenmilieu.wur.nl.

Statutory Research Tasks Unit for Nature & the Environment
(**Wettelijke Onderzoekstaken Natuur & Milieu)**
Postbus 47, 6700 AA Wageningen
Tel: (0317) 47 78 44; Fax: (0317) 42 49 88; e-mail: info@npb-wageningen.nl;
Internet: www.wotnatuurenmilieu.wur.nl

Contents

Abstract and Acknowledgements

Abstract
ASSA is a public-domain open-source library of algorithms for stochastic sensitivity analysis in ANSI C. It is a documented collection of basic and more sophisticated algorithms in that field.

Acknowledgements
The research enabling the construction of ASSA has been performed in the framework of strategic research of Biometris and in various consultation projects. The actual writing up was supported by WOT N&M, the Dutch Statutory Research Task Unit for Nature & the Environment at Wageningen University and Research Centre. The stimulating conversations with Harm Houweling of that office constituted vital support. I am grateful to Jacques Withagen of Biometris for code-checking and comparison of ASSA results with results from other software.

1 Introduction

In stochastic sensitivity analysis (SSA), the imperfectly known inputs of a mathematical model are stochastic variables. The joint probability distribution of these inputs is supposed to be known at the start of the analysis, so quite some hard work has to be done in advance. The analysis assesses the effects on model output caused by the uncertainty in distinct inputs. These inputs may comprise parameters, exogenous variables, initial conditions and so on.

Instead of the effect of distinct *individual* inputs, one may just as well analyse the effect of distinct *groups* of inputs.

The study of the combined effect of all uncertain inputs is often called *uncertainty analysis*.

The adjective 'stochastic' in SSA is added to distinguish the subject from all kinds of deterministic sensitivity analysis, but very often SSA is plainly called sensitivity analysis. Some authors restrict the term SSA to stochastic differential equation models, but we will use the term in the broader sense just described.

In the type of Monte Carlo analysis discussed here, one constructs a random sample from the joint distribution of the inputs; the model is run for each sampled input vector (sometimes very computer-intensive); after that, one looks what the input variation does to the output. Usually, the computational effort is nearly proportional to the number of model runs, so efficiency is sought for in minimisation of the required number of runs (and nowhere else). The point is that the results of the analysis become more accurate as the number of runs increases.

Various software products exist for SSA. Saltelli *et al*. (2000) contains an overview of software available in the year 2000. The software packages mentioned there are closed in the sense that you can hardly change or add components. The section in the same book on generic algorithms is still far from complete. Thus, there does not seem to exist a fairly complete, coherent, and documented collection of algorithms for SSA in a basic programming language like C or Fortran. The ASSA project has the purpose to begin filling this gap.

The collection is available in the public domain, in such a form that everyone can use the software freely. It is hoped that users will suggest improvements or additions. The long-term goal is a collection of documented algorithms in the spirit of the famous series of Numerical Recipes (e.g. Press *et al*., 1992), but with a slightly different legal status.

Model builders should be enabled to incorporate the algorithms into their own software, for instance in order to accompany model statements with an indication of inaccuracy due to input uncertainty. Another application is inclusion of SSA algorithms into frameworks for building, coupling and analysing models.

The language used is ANSI C, written in such a style that translation into another basic programming language should not give rise to serious problems.

At present, ASSA consists mainly of conventional algorithms for sensitivity analysis. Apart from auxiliary routines, the algorithms can be divided into:

- algorithms for constructing model input samples; and
- algorithms for analysing the corresponding model output samples.

All sensitivity analyses in the current version of ASSA are variance-based.

The long-term goals are:

- a gradually improving and extending collection of basic and more advanced algorithms for SSA, leading to a moderate form of standardisation;
- uniform description of these algorithms via C-programs;
- a form of publication inviting comments and additions, while enabling flexible use of the algorithms.

This report is a manual, not a theoretical exposition. Appendix 1 is an exception: it describes some details that we could not easily find in the literature for sensitivity analysis. For an overview of the theory, see for instance Saltelli *et al*. 2000). For theoretical details, see the literature cited.

2 Overview of algorithms

2.1 Input generators

Random generators are used to construct a sample from the distribution of uncertain parameters or other model inputs. In the present version, the statistical properties of the inputs can be described in terms of their grade correlation (often loosely called rank correlation). Each of the individual variables is defined through the type and parameters of its distribution.

There are two basic random generators: uniform(0,1) and multinormal(μ, Σ). For the rest, the drawing of random samples is done in two steps.

The first step draws a sample from the k-dimensional unit-hypercube. Each of the k variables thus sampled is more or less randomly and more or less uniformly distributed over the interval (0,1). Some examples: independent; dependent with given grade correlation; latin hypercube (McKay *et al*., 1979; Iman and Conover, 1980; Stein, 1987; Owen, 1992); latin hypercube with forced rank correlation (Iman and Conover, 1982; Helton and Davis, 2002); and – by way of example – a systematic sample constructed from a saturated main-effect design.

The next step transforms these (0,1) variables into variables with the required distribution. The distributions currently available are: uniform, triangular, normal, log-normal, beta and gamma. Auxiliary routines are supplied to derive the standard parameters of distributions from information about means and variances, or about quantiles.

2.2 Analysis

In ASSA's present version, all sensitivity analyses are variance-based, i.e. they perform some kind of analysis of variance on the model output. During the 1990's there seems to have grown consensus that this form of sensitivity analysis is very adequate for most purposes. The algorithms provide the possibility to estimate the variance contributions of *groups* of inputs, which often facilitates the interpretation of the results, especially when variables from different groups are stochastically independent. There is an algorithm for the most common form of sensitivity analysis: the one based on linear regression. We give a simple example of a regression-free sensitivity analysis. It is shown how one can calculate a *bootstrap confidence* interval for the sensitivity estimates from these analyses if the input sample consists of independent draws from the input's uncertainty distribution.

An analysis based on spline-regression is still on the list of wishes. An algorithm for winding stairs analysis (Jansen *et al*. 1994; Jansen, 1996) is in the planning.

For the time being, only two simple test functions are included, from which sensitivity properties can be calculated analytically.

2.3 Auxiliary routines

ASSA contains routines to summarise the statistical properties of a sample: variance matrix, correlation matrix, mean, variance, median and rank-correlation. There is an algorithm to check if a symmetric matrix is positive definite. Graphical routines are not included.

3 Supporting software required

The current version of ASSA frequently uses algorithms from Numerical Recipes in C (NRC; Press *et al*., 1992). Thus, you may only use this version of ASSA in applications where you are entitled to use the algorithms from Numerical Recipes. The NRC procedures used for ASSA serve mainly to allocate and free memory space for vectors and matrices, to generate uniform random numbers, and to calculate special functions relating to probability distributions.

The NRC utility files `nrutil.h` and `nrutil.c` are used primarily because their contents are called by the NRC routines used in ASSA, but also to allocate and free memory space for vectors and matrices. For example the procedures `vector, free_vector, matrix, free_matrix, ivector, free_ivector`.

The NRC procedures used are: `ran1, betacf, betai, gammln, gammp, gasdev, gcf, gser, indexx`. These and all other NRC procedures are prototyped in `nr.h`.

We used the ANSI-C version 2.10 of numerical recipes. If you have an older version 2.x it will probably work just as well, but you can obtain an upgrade via www.nr.com.

Side effects

ASSA inherits the ease-of-use of NRC's vector and matrix routines, but also their risk-of-use. A programming error like offering a 4-by-100 matrix to a procedure that expects a 100-by-4 matrix will go unnoticed by the system. You may call yourself lucky if such programming errors cause a halting of the execution by the operating system. The latter would happen in the opposite case of offering a 100-by-4 matrix to a procedure that expects a 4-by-100 matrix. Remember that a C-programmer is supposed to know what he is doing, and carefully read the NRC sections on vectors and matrices.

By the inclusion of `nrutil.h` and `nrutil.c`, NRC constructs like `DMAX` or `DSQR` were available, but they have not been used in the recipes themselves; possibly in examples. With some compilers, these NRC constructs cause warnings that variables have been declared but not been used. To suppress such warnings, you might adapt the NRC files, but we have chosen to use NRC as it is.

4 Conventions

4.1 Data types

All floating point numbers are of the double type, all integers are of the (signed) long type. Even the function `main` is of the long type. The NRC procedures are adapted to this convention by brute force, namely by stating in the header file `assa.h`

```
/* include NRC ANSI prototypes */
#define float double
#define unsigned
#define int long
#ifndef ANSI
  #define ANSI
#endif
#include "nrutil.h"
#include "nr.h"
#undef float
#undef unsigned
#undef int
```

Similarly, in the file `assa.c`, we state

```
/* include NRC ANSI routines */
#define float double
#define unsigned
#define int long
#ifndef ANSI
  #define ANSI
#endif
#include "nrutil.c"
#include "ran1.c"
#include "gasdev.c"
#include "betacf.c"
#include "betai.c"
#include "gammln.c"
#include "gammp.c"
#include "gcf.c"
#include "gser.c"
#include "indexx.c"
#undef float
#undef unsigned
#undef int
```

The substitution of `long` for `int` would not generate correct C if one uses phrases like `long int`, but these do not occur in the NRC-files.

Full exploitation of the change from float to double in a numerical procedure, would require that several precision parameters of the procedure are adapted. If you really want to make such adaptations in the NRC routines used, you will find useful hints in the C++ version of Numerical Recipes, which also has the double type as default (Press et al., 2002; Appendix C).

A type boolean is made available by the next statement in `assa.h`

```
typedef enum {false, true} boolean
```

4.2 Special constants

Two special constants are defined in `assa.h` through

```
#define ASSA_MISSING -99999.
#define ASSA_EPS 3.0e-7
```

4.3 Names for frequently recurring items

N	sample size
k	number of input variables
X	k-by-N input sample matrix
U	k-by-N matrix with columns in the unit hypercube
y	N-vector of model output corresponding to X
μ	mean (called `mu` in the code)
σ^2	variance (called `sigmasq` in code); 'v' is also used
V	variance-covariance matrix
C	correlation matrix

4.4 Symmetric matrices

In ASSA, symmetric double matrices – usually variance matrices – are allocated as square matrices, but only the left lower triangle is used. Thus, if V is a k-by-k variance matrix, you only need to assign values to the elements V_{ij} with $j \le i \le k$.

5 Elementary procedures

Some elementary procedures are given, without much explanation. The procedures are required occasionally, but they are no typical ASSA-material. We hope that it is sufficiently obvious what these procedures have to do, and how they do it. If not, please read the C-code.

5.1 Summary statistics

Procedure `summary` calculates mean, variance, minimum, maximum and median of N-vector `x` = $(x_1 \ldots x_N)$. Pointers to doubles where the results must be stored are given as arguments.

```
void summary(double *x, long N,
        double *mean, double *var,
        double *min, double *max, double *med)
```

Procedure `msummary` calculates means, variances, minima, maxima, medians and k-by-k variance matrix of k-by-N sample matrix X.

```
void msummary(double **X, long k, long N,
        double *mean, double *var, double *min, double *max,
        double *med, double **V)
```

The results of `msummary` are stored in existing k-vectors mean, var, min, max, med, and in existing k-by-k matrix V. Only the left lower triangle of V is filled, the rest is left unchanged.

Function `quantile` gets an N-vector x, which is treated as a sample from a continuous distribution F. It calculates the '`prob`-quantile': an estimate of the value ξ such that the $F(\xi) =$ `prob`.

```
double quantile(double prob, double *x, long N)
```

The next procedure calculates the k-by-k variance matrix V of k-by-N matrix X, which is treated as a sample of k N-vectors.

```
void calc_varcov(double **X, double **V, long k, long N)
```

Void `calc_corr` treats k-by-k matrix V as a variance-matrix, and calculates the corresponding k-by-k correlation matrix C. Only the lower triangular part of V is taken into account, and only the lower triangle of C is assigned.

```
void calc_corr(double **C, double **V, long k)
```

Procedure `calc_rank` calculates the ranks of N-vector x. If all x-values are different, the lowest x-value gets rank 1 and the highest rank N. Duplicate or multiplicate x-values get different ranks.

```
void calc_rank(double *x, double *rank, long N)
```

5.2 Matrix operations

Procedure `cholesky` calculates the lower triangular square root, `L` say, of a positive definite k-by-k symmetric matrix `S`, in the sense that $L.L^T = S$, where L^T is the transposed of `L`. The k-by-k matrix `L` must have been declared in advance. The above-diagonal part of `L` is filled with zero's. As stated earlier, only the left-lower triangle of symmetric matrix `S` must have values, the upper part is ignored.

```
void cholesky(double **S, double **L, long k)
```

The procedure is used in ASSA for the generation of multinormal random variables.

Procedure `cholesky` is also used as a step in the inversion of a positive definite matrix; it works as follows. Routine `lowinv` calculates the inverse of k-by-k lower-triangular matrix `L` and puts the result in existing n-by-n lower triangular matrix `LINV`. The above-diagonal part of `L` is ignored; the above-diagonal part of `LINV` is filled with zero's.

```
void lowinv(double **L, double **LINV, long k)
```

A variance matrix, or any other positive-definite symmetric matrix, can be simply inverted by Cholesky decomposition and inversion of the ensuing lower-triangular matrix. Such an inversion will be applied in the method of Iman and Conover to construct input samples with a prescribed rank correlation (see chapter: Generators in the Unitcube).

Boolean function `posdef` checks if k-by-k matrix `S` is positive definite by going through the steps made by `cholesky`, but instead of issuing an error message, it returns `false` if `S` is not positive definite (and otherwise `true`).

```
boolean  posdef(double **S, long k).
```

5.3 Utilities

Procedure `error` is used to end program execution, after issuing a message on the standard error channel `stderr`.

```
void error(char text[]).
```

It may be used like this:

```
error("cholesky: input matrix not positive definite").
```

On the other hand, void `warning` only issues a message and allows the program to continue:

```
void warning(char text[]).
```

6 Basic random generators

The uniform(0,1) random generator in ASSA is `ran1` from NRC. It is the heart of all ASSA's random generators. In a program using ASSA, the seed is a `long int`, `seed` say, that should be set to some negative value once, before any of the random generators are called. This will initialise *all* random generators, since all of them are based on `ran1`. Every random generator used afterwards should use a *pointer* to this one and only seed as argument. There seems to be no need to reinitialise the generators since `ran1` has a very long cycle, but if you really want to reinitialise the random generators within a program, assign a negative value to `seed`.

A uniform random `long int` from the set {1, 2, ... N} can be drawn via

```
long any(long N, long *seed)
```

which rounds `N*ran1(*seed)` to the nearest larger integer.

You can fill an existing long-vector x of size N with a random permutation of the vector (1, 2, ... N) by applying

```
void perm(long *x, long N, long *seed)
```

This function draws any number out of 1...N, then from the remaining N-1 numbers, and so on. The main task of the procedure is a bookkeeping of the numbers that have not yet been drawn. The permutation is returned in long-vector x.

You can fill an existing long-vector x of size N with a bootstrap sample of the vector (1, 2, ... N) by applying

```
void boot(long *x, long N, long *seed)
```

This function draws N times any number out of 1...N. The bootstrap sample is returned in long-vector x. The procedure can be used as follows: let y be an N-vector, then y[x[1]], y[x[2]] ... y[x[N]] constitutes a bootstrap sample from y.

The standard normal random generator, having mean $\mu=0$ and variance $\sigma^2=1$, is NRC's `gasdev`, which internally calls `ran1` as uniform(0,1) random generator (so it uses the same seed).

ASSA's multivariate normal generator

```
void mnor_mat(double **X, double *mean, double **V,
              long k, long N, long *seed)
```

fills an existing k-by-N double matrix X. The N columns of the matrix constitute N independent draws from a k-dimensional normal distribution with mean vector mean and with k-by-k variance matrix V. Here, as everywhere in ASSA, k-by-k symmetric double matrices are allocated as full k-by-k square matrices, but only the lower triangle is used (see the Conventions section). The function checks whether the matrix V is positive definite. If not, a fatal error message is issued. The procedure is based on NRC's cholesky decomposition. The internally used boolean function `posdef` checks if V is positive definite by preliminary going through the steps to be made by `cholesky`.

Example

In this example, a mean vector 0, and a variance matrix V are constructed for 8 variables (all variances are 1, so the matrix also happens to be a correlation matrix). The covariances are 0 except V[3][2] and V[5][4], which are 0.75. A multinormal sample of size 1000 is drawn, with some seed, using `mnor_mat`. The properties of resulting sample matrix X are calculated using `msummary`.

```
#include "assa.h"

int main()
{
   long seed=-290405, N=1000, k=8, i, j;
   double **V, **X, *mean, *var, *min, *max, *med;
   mean = vector(1, k);
   var  = vector(1, k);
   min  = vector(1, k);
   max  = vector(1, k);
   med  = vector(1, k);
   V = matrix(1, k, 1, k);
   X = matrix(1, k, 1, N);
   for (i=1; i<=k; i++) {
      mean[i] = 0;
      for (j=1; j<=k; j++) V[i][j] = (i==j)? 1: 0;
   }
   V[3][2] = 0.75; V[5][4] = 0.75;
   mnor_mat(X, mean, V, k, N, &seed);
   msummary(X, k, N, mean, var, min, max, med, V);
   for (i=1; i<=k; i++) {
      for (j=1; j<=i; j++) printf("%6.2f ", V[i][j]);
      printf("\n");
   }
   printf("\n\n");
   printf("     j    mean     var     min     max     med\n");
   for (j=1; j<=k; j++) printf(
        "%6ld  %6.3f  %6.3f  %6.3f  %6.3f  %6.3f\n",
            j, mean[j], var[j], min[j], max[j], med[j]);
   return 1;
}

#include "assa.c"
```

7 Probabiliy distributions and related functions

Assa contains cumulative distributions and related special functions for the *normal, lognormal, triangular, gamma and beta* distributions. Firstly, there are the cumulative distribution functions. The inverses of the cumulative distribution functions can be used to transform uniform(0,1) random variables into random variables of other types. The inverse F^{-1} of some *continuous* cumulative distribution function F, has the property that $F^{-1}(F(x)) = x$. It is well-known that for a random variable u with a uniform(0,1) distribution, the transformation $F^{-1}(u)$ is random with distribution function F. Some additional functions translate information about means and variances, or about a pair of quantiles, into the usual parameters of the distributions.

7.1 Normal distribution

The cumulative *standard* normal distribution, with mean 0 and variance 1, is implemented in the function

```
double pnormal(double x)
```

Argument `x` may have any real value; the function returns a double in the interval (0,1). The code for `pnormal` is based on an approximation formula in Abramowitz and Stegun (1965). The cumulative normal distribution with mean μ and variance σ^2 can be obtained through

```
pnormal((x-mu)/sigma)
```

The inverse of `pnormal`, the function

```
double invnormal(double p)
```

delivers the quantity, say `x`, such that `pnormal(x)=p`. If `u` is a uniform(0,1) random variable, then `invnormal(u)` is a standard normal variable, with mean 0 and variance 1. The code for `invnormal` is based on an approximation formula in Abramowitz and Stegun (1965). A normal variable with mean μ and variance σ^2 is obtained by

```
mu + sigma * invnormal(u)
```

with `u` uniform(0,1).

The next function translates information about a pair of quantiles of the normal distribution into the values of the mean and variance of that distribution.

```
void q2m_normal(double *mean, double *variance,
                double p1, double p2, double q1, double q2)
```

in which `q2m` is shorthand for 'quantiles to moments'. The quantile information has the following meaning. Let x denote the random variable; then p1 = P(x<q1) and p2 = P(x<q2).

Example

The following example calculates and prints parameters mu and sigma of a normal distribution with 10% point 6 and 90% point 7. Next it constructs and prints a vector of 1000 independent draws from this distribution.

```
#include "assa.h"
int main()
{
   long seed=-210205, N=1000, i;
   double *z, mu, sigmasq, sigma, p1, p2, q1, q2;
   z = vector(1, N);
   p1 = 0.1; p2 = 0.9;
   q1 = 6;   q2 = 7;
   q2m_normal(&mu, &sigmasq, p1, p2, q1, q2);
   sigma = sqrt(sigmasq);
   printf("mu = %f sigma= %f\n", mu, sigma);
   for (i=1; i<=N; i++){
      z[i] = mu + sigma*invnormal(ran1(&seed));
      printf("%f\n", z[i]);
   }
   return 1;
}
#include "assa.c"
```

7.2 Lognormal distribution

For the lognormal distribution a small number of functions is available in ASSA. The first one translates information about a pair of quantiles of the lognormal distribution into the values of the mean and variance of that distribution.

```
void q2m_lognormal(double *mean, double *variance,
                double p1, double p2, double q1, double q2)
```

An error message follows if the information about the quantiles is inconsistent.

The next function translates the mean and variance of a lognormal distribution into the mean `mu` and the standard deviation `sigma` of the underlying normal distribution.

```
void m2p_lognormal(double *mu, double *sigma,
                   double mean, double variance)
```

Thus, if `z` is normal with mean `mu` and standard deviation `sigma`, exp(z) is lognormal with mean and variance.

Example

This example shows how to draw a size-1000 sample from a lognormal distribution with mean 10 and variance 1. The mean, variance, minimum, maximum and median of the sample are calculated and printed.

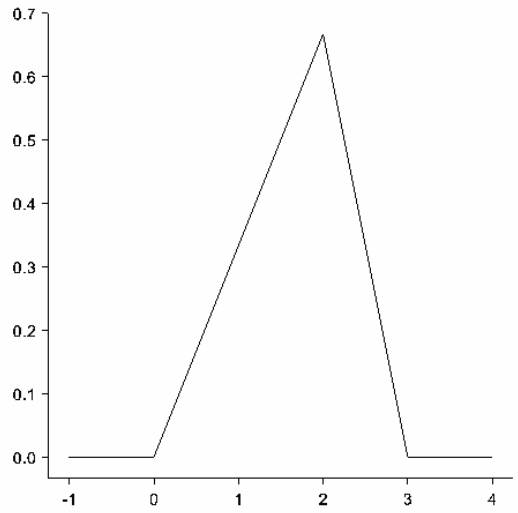
```
#include "assa.h"
int main()
{
```

```
    long seed=-782393, N=1000, i;
    double *x, mean=10, variance=1, mu, sigma,
            smean, svar, smin, smax, smed;
    x = vector(1, N);
    m2p_lognormal(&mu, &sigma, mean, variance);
    for (i=1; i<=N; i++) x[i] = exp(mu + sigma * gasdev(&seed));
    summary(x, N, &smean, &svar, &smin, &smax, &smed);
    printf("%5.2f %5.2f %5.2f %5.2f %5.2f",
            smean, svar, smin, smax, smed);
    return 1;
}
#include "assa.c"
```

Note that there are two stochastically equivalent ways to draw a standard normal random number, by `gasdev(&seed)` and by `invnormal(ran1(&seed))`.

7.3 Triangular distribution

The triangular distribution is included since its properties are so easy to understand, and since it provides a useful refinement of the uniform distribution as a rough and intuitive characterization of a random variable. It is parameterised by the locations of its lower bound, its top and its upper bound. For example, the next figure shows the density function of a triangular distribution with lower bound 0, top 2 and upper bound 3.



The cumulative triangular distribution function has the prototype

```
double ptriang(double x, double low, double top, double hig).
```

Argument `x` may have any real value; the other arguments must satisfy `low < hig` and `low ≤ top ≤ hig`; the function returns a double in the interval (0,1); it returns 0 if `x≤low`, and 1 if `x≥hig`.

The prototype of the inverse triangular distribution is given by

```
double invtriang(double p, double low, double top, double hig).
```

7.4 Gamma distribution

The gamma distribution is useful to describe positive random variables. Its density function has two parameters, 'shape' parameter a>0 and 'scale' parameter b>0:

$$f(x) = x^{a-1} e^{-x/b} b^{-a} / \Gamma(a)$$

(We explicitly mention this density function because the gamma distribution is not always parameterised in the same manner.) Its mean is $\mu = a\ b$; its variance $\sigma^2 = a\ b^2$.

The cumulative gamma distribution function is calculated by the function

```
double pgamma(double a, double b, double x)
```

the function returns a double in the interval (0,1). The code for this function uses NRC's incomplete gamma function.

The prototype of the inverse gamma(a, b) distribution is

```
double invgamma(double p, double a, double b)
```

This inverse is calculated via bisection on the cumulative distribution function `pgamma`. The precision of the result will probably suffice for most purposes, but you may wish to alter the code for more precision.

The next procedure translates the values of the mean and variance of a gamma distribution into the corresponding values of the parameters a and b.

```
void m2p_gamma(double *a, double *b, double mean, double variance)
```

The prefix `m2p` is shorthand for 'moments to parameters'.

Example

This example shows how to draw a size-1000 sample from a gamma distribution with mean 10 and variance 1. The mean, variance, minimum, maximum and median of the sample are calculated and printed. Note that the specification of the distribution is the same as in a previous example for the lognormal distribution. In such a situation, there seem to be no strong arguments to choose for the lognormal or the gamma distribution.

```
#include "assa.h"
int main()
{
   long seed=-784823, N=1000, i;
   double *x, mean=10, variance=1, a, b,
           smean, svar, smin, smax, smed;
   x = vector(1, N);
   m2p_gamma(&a, &b, mean, variance);
   for (i=1; i<=N; i++) x[i] = invgamma(ran1(&seed), a, b);
   summary(x, N, &smean, &svar, &smin, &smax, &smed);
   printf("%5.2f %5.2f %5.2f %5.2f %5.2f",
           smean, svar, smin, smax, smed);
   return 1;
}
#include "assa.c"
```

7.5 Beta distribution

The beta distribution describes a random variable with values between 0 and 1. If the uniform and the triangular distribution are not satisfying to describe a random variable that is bounded both from above and from below, the beta distribution may be useful. It is parameterised by two parameters a>0 and b>0. The density function has the form

$$f(x) = x^{a-1} (1-x)^{b-1} \quad / B(a, b)$$

Its mean is $\mu = a / (a+b)$; its variance is $\sigma^2 = a\, b / \{(a+b)^2 (a+b+1)\}$. The cumulative distribution function is realised by the routine

```
double pbeta(double a, double b, double x)
```

which returns a double in the interval (0,1). The code for this function uses NRC's incomplete beta function.

The prototype of the inverse beta(a, b) distribution is

```
double invbeta(double p, double a, double b)
```

This inverse is calculated via bisection on the cumulative distribution function pbeta. The precision of the result will probably suffice for most purposes, but you may wish to alter the code for more precision.

The next function translates the mean and variance of a beta distribution into the parameters a and b:

```
void m2p_beta(double *a, double *b, double mean, double variance)
```

This function produces an error message if the mean lies outside the interval (0,1), or if the variance is impossibly large.

8 Generators in the unit hypercube

All unit hypercube, UHC, generators to be mentioned in this chapter produce a k-by-N matrix with values strictly between 0 and 1 (i.e. without attaining the boundaries). The procedures for these generators have 'uhc' in their name. Each of the N columns of U can be viewed as a point within a unit hypercube of dimension k. Resulting matrix U will be used as a sample of N points in the k-unit-hypercube. The nature of the sample ranges from totally random to almost totally deterministic. Invariably, the individual rows of the matrix contain values that are uniformly distributed over the interval (0,1) excluding 0 and 1. The k-by-N matrix U should exist before the procedures are called.

We start with the most elementary – ordinary random – generator in the unit hypercube, with this prototype:

```
void uhc_basic(double **U, long k, long N, long *seed)
```

It fills existing k-by-N matrix U with N independent draws from the uniform distribution on the k-unit hypercube. Thus each element of U is uniform(0,1) independent from all others.

The next procedure produces N correlated, marginally uniform draws in the k-dimensional unit-cube.

```
void uhc_corr(double **U, double **C, long k, long N, long *seed)
```

Existing k-by-N matrix U is filled with N independent draws from a correlated marginally uniform distribution on the k-dimensional unit-hypercube. Within a draw, each of the k elements are from a uniform(0,1) distribution, while the elements have correlation matrix `C`, or very nearly so. These properties concern the distribution from which is drawn, the sample properties will deviate, especially in small samples.
The procedure is based on a remarkable property of the multinormal distribution, namely the near-equality of its Pearson and rank correlation (grade correlation to be very precise). Let k-vector z have a multinormal distribution with mean vector 0, and *covariance* matrix `C`. Thus, each component z_i of z has a standard normal distribution. Let Φ denote the standard normal distribution function. Then $\Phi(z_i)$ has a uniform(0,1) distribution, since the distribution function of any continuous random variable is uniform(0,1). But the remarkable fact is that the correlation between $\Phi(z_i)$ and $\Phi(z_j)$ is very nearly equal to `C`[i][j]: the difference is 0.018 at most (see Appendix 1 for details). With `mnor_mat()` the procedure fills a matrix with N independent multinormal(0, C) drawings. Next it applies the function Φ to them.

First best?
In several aspects, the first two generators mentioned above are also the best ones. Their simplicity is an advantage when you have to explain the procedure. Moreover, since they produce N *mutually independent* draws from the k-unitcube, you can easily enlarge the sample, and you have special possibilities to evaluate the precision of the results of the ensuing sensitivity analysis. The results of that analysis are random, so there will be a question about their accuracy. Only with a sample consisting of independent draws, you can say something about the precision on the basis of *one* sample. With the other sampling methods that will follow in this chapter, you will need to make *multiple* samples and compare and combine the analysis results of these

samples in order to assess result precision. So an independent sample is always the best choice? Not under all circumstances. Although you can calculate the precision, you will know in advance that the precision is lower than the, harder to assess, precision of the procedures to follow. Thus you have a choice problem if you want to calculate precision of results – as you should.

The next procedure performs basic latin hypercube sampling, which takes care that the marginal *sample* distribution of each variable is highly uniform (McKay, Beckman and Conover, 1979; Iman and Conover, 1980).

```
void luhc_basic(double **U, long k, long N, long *seed)
```

The k-by-N matrix U gets N latin hypercube draws from the distribution on the k-unit-hypercube; within one draw, the elements are independently uniform(0,1). But the whole N-sample U[i][·] of the i-th variable contains precisely 1 value in each of the intervals (0, 1/N), (1/N, 2/N) ... ((N-1)/N, 1).

For the construction of the UHC generators, ASSA has two auxiliary rank matrix generators. These auxiliary generators are called by the other generators. Here is the first:

```
void rank_mat(long **R, double **C, long k, long N, long *seed)
```

Existing matrix R will be filled with a random k-by-N rank matrix for a given population rank correlation matrix `C`. The procedure is based on the near-equality of the Pearson and rank correlation of the multinormal distribution (see above). It fills a matrix with N independent multinormal(0, C) drawings, and then calculates the rank per row.

A correlated latin hypercube sample in the unit hypercube can be obtained via

```
void luhc_corr(double **U, double **C, long k, long N, long *seed)
```

By this procedure, existing k-by-N matrix U gets N latin hypercube draws in the k-dimensional unit-hypercube; within one draw, each element is uniform(0,1). The sample's rank correlation is randomly drawn given population rank correlation matrix C using `rank_mat`.

Just as one can force highly uniform marginal sample distributions by drawing a latin hypercube instead of an ordinary random sample, one may wish to force the sample's rank correlation matrix, to be close to a desired correlation matrix. A set of ranks with such a forced correlation can be obtained via the second auxiliary rank matrix generator of ASSA:

```
void iman_rank_mat(long **R, double **C, long k, long N, long *seed)
```

By this procedure, the rows of existing k-by-N integer matrix R will hold vectors of ranks with correlation-matrix very close to given C. Exact equality cannot be attained in general, since correlations between two permutations of the vector (1...N) cannot attain any value between -1 and 1. The procedure is an accurate implementation of the method of Iman and Conover for the construction of such a matrix as described by Helton and Davis (Iman and Conover, 1982; Helton and Davis, 2002). Internally, the procedure uses `lowinv` to calculate the inverse of correlation matrix C.

The next procedure produces a latin hypercube sample with forced sample rank-correlation

```
void luhc_iman(double **U, double **C, long k, long N, long *seed)
```

By this procedure, existing k-by-N matrix U is filled with N latin hypercube draws from the distribution on the k-unit-hypercube. Within a draw, each element is uniform (0,1). The correlation matrix of the rows of U closely resembles given C. The procedure is an accurate implementation of the method of Iman and Conover as described by Helton and Davis, and uses `iman_rank_mat` (Iman and Conover, 1982; Helton and Davis, 2002).

A very high degree of non-correlation may be reached by the following even more systematic way of sampling. The procedure is based on a so-called SAturated MAin effect fractional factorial design (cf. Dey, 1985, Section 2.4.1). The dimensions of the matrix U must have special values for this purpose. Let `prime` be a prime number (i.e. ≥2) and let `power` be a `long int` ≥2. The procedure will fill an existing k-by-N matrix U with $N = prime^{power}$ columns and $k = (prime^{power}-1) / (prime-1)$ rows (the latter division always produces an integer).

```
void uhc_sama(double **U, long prime, long power, long *seed)
```

U will contain highly-restricted random draws in the k-unit-hypercube. The correlation between the rows of U is very close to 0. The original deterministic design, constructs $(prime^{power}-1)/(prime-1)$ orthogonal factors of length $N = prime^{power}$ at `prime` levels `0...prime-1`. Next the factors are randomised, augmented with a homogeneous(0,1) term and divided by `prime`. Of course the user need not use all the rows of matrix U in his subsequent calculations, but when uhc_sama is called, a matrix U of the dimensions mentioned should exist.

Example

```
#include "assa.h"
int main()
{
   long seed=-140105, k1=3, N1=10,
        prime=3, power=2, k2=4, N2=9;
   long i,j;
   double **U1, **U2;
   U1 = matrix(1,k1,1,N1);
   U2 = matrix(1,k2,1,N2);
   uhc_basic(U1, k1, N1, &seed);
   for (j=1; j<=N1; j++) {
      for (i=1; i<=k1; i++) printf(" %10.4f", U1[i][j]);
      printf("\n");
   }
   printf(":\n");
   luhc_basic(U1, k1, N1, &seed);
   for (j=1; j<=N1; j++) {
      for (i=1; i<=k1; i++) printf(" %10.4f", U1[i][j]);
      printf("\n");
   }
   printf(":\n");
   uhc_sama(U2, prime, power, &seed);
   for (j=1; j<=N2; j++) {
      for (i=1; i<=k2; i++) printf(" %10.4f", U2[i][j]);
      printf("\n");
   }
```

```
printf(":\n");
   return 1;
}
#include "assa.c"
```

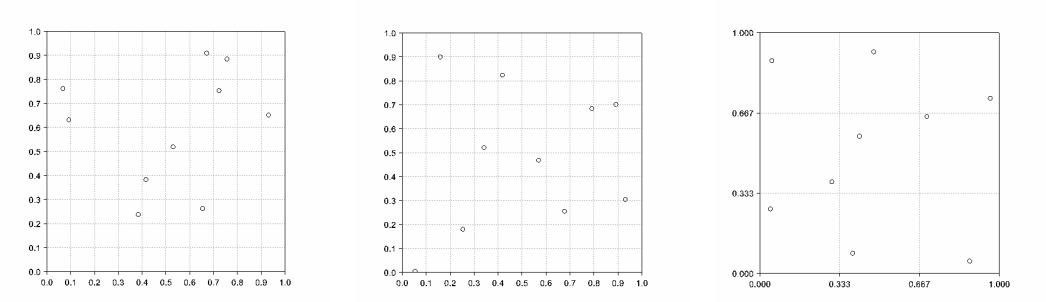


Figure. Three – for clarity somewhat small – samples from uhc_basic(.,3,10,.), luhc_basic(.,3,10,.) and uhc_sama(.,3,2,.). Graphs of two rows from the sample matrix against one-another. In the latin hypercube sample each of the ten horizontal slices contains one point, while the same applies to the vertical slices. In the sama sample, each combination of a horizontal and a vertical slice contains one point.

Which one to choose?

There remains a problem of choice, already with the small number of alternatives presented above. Primarily the choice depends on the uncertainty distribution of the k inputs studied. The dependence or independence of the k components dictates whether to choose a unitcube sampler with or without correlation.

If model runs are inexpensive, if you like simplicity, and if you know how to use resampling methods to assess result precision of the sensitivity analysis, you might use `uhc_basic` or `uhc_corr`. You can enlarge the sample if the precision is unsatisfactory, without discarding outcomes of model runs made before.

If model runs are expensive, replicated latin hypercube sampling with forced correlation via `luhc_iman` is probably the most attractive candidate. You can force non-correlation if the inputs are independent. Three replicates will give only a very rough impression of result precision. An great advantage is that the Iman-Conover method is well-known. In case of non-correlation, you might consider to use the more exotic `uhc_sama,` which is probably even more accurate than Iman-Conover.

Example

With the method of Iman and Conover for correlated latin hypercube samples, the next code produces a sample of size 1000 of 3 correlated random variables. The intermediate unitcube sample is stored in matrix U. Next the three variables are transformed into a normal(10, 1) variable, a gamma(1, 2) variable, and a beta(2, 3) variable, just to show how the inverse distribution functions can be applied. For a realistic application, the parameters of the distributions should be carefully determined.

```
#include "assa.h"
int main()
{
```

```
   long seed=-030205, k=3, N=1000;
   long i,j;
   double **U, **X, **C;
   U = matrix(1, k, 1, N);
   X = matrix(1, k, 1, N);
   C = matrix(1, k, 1, k);

   C[1][1]=1;
   C[2][1]=0; C[2][2]=1;
   C[3][1]=0.5; C[3][2]=0; C[3][3]=1;
   luhc_iman(U, C, k, N, &seed);
   for (i=1; i<=N; i++){
      X[1][i] = 10 + 1*invnormal(U[1][i]);
      X[2][i] = invgamma(U[2][i], 1, 2);
      X[3][i] = invbeta(U[3][i], 2, 3);
   }
   for (i=1; i<=N; i++){
      for (j=1; j<=k; j++) printf("%10.4f", X[j][i]);
      printf("\n");
   }
   return 1;
}
#include "assa.c"
```

Analysis of the sample printed by the above code, shows that the means and variances of the sample of the three variables are respectively

```
mean:          9.9996    1.9999    0.4000
variance:      1.0003    3.9952    0.0400
```

You can check that these sample values are close to the distribution values. The covariance matrix of the three rows of intermediate unitcube sample U is equal to

```
    0.0834
    0.0003    0.0834
    0.0405    0.0006    0.0834
```

Which shows that the variance of each intermediate u-variable is very close indeed to the expected value 1/12 of a uniform(0, 1) variable. The sample correlation between the uncorrelated variables is very close to 0, while the sample correlation between the unitcube first and third variable is equal to 0.0405/0.0834=0.4856: quite close to the value 0.5 of the target rank correlation C[3][1]. If the function `luhc_iman(U, C, k, N, &seed)` would be replaced by `uhc_corr(U, C, k, N, &seed)`, the sample values would often be further from the target values.

9 Variance components

In the first quarter of the 20-th century, the word 'variance', and the technique of analysis of variance components have been introduced in statistical genetics. The purpose was to assess the relative influences of factors like parents and environment on some property of a plant or an animal. The variance was chosen because the variances contributed by independent causes of variability nicely sum up to the total variance (e.g. Fisher Box, 1978, p 53). Later, analysis of variance components experiments were conducted to assess the prospects of a genetic selection program (e.g. Comstock and Robinson, 1948). If, for instance, environment contributes much more than parents to the variance of a property, little can be gained through genetic selection.

Application of analysis of variance components to model output, stems from the 1990s (Sobol, 1990; Jansen, Rossing and Daamen, 1994; Sobol, 1995; McKay, 1996; Saltelli, Tarantola and Chan, 1999; Jansen, 1999; Saltelli, Chan and Scott, 2000, Ch 8). The new purpose of the analysis of variance components was to pinpoint major sources of model output uncertainty and evaluate the prospects to reduce this uncertainty by gaining more accurate information about some inputs (or by better control of some inputs). Some model experiments are almost the same as the old genetics experiments. The additivity of the variance contributions of independent components remains the major reason to chose variance as measure of uncertainty. A second advantage is that properties of the variance relevant for sensitivity analysis can often be found in the classical statistical literature.

The type of sensitivity analysis sketched above is called variance-based because uncertainties and uncertainty contributions are expressed as variances and variance components. Some of these variance-based sensitivity analyses are non-parametric, namely when they do not rely on an estimated parametric relationship between input and output. The analysis of variance is a formal elaboration of the intuitive idea that a factor is important if the model output studied changes much when that factor assumes new random values while the other factors remain the same. The statistical methods have remained essentially the same with the change of application from genetics to uncertainty analysis, but the experimental designs differ somewhat, mainly because of the practical restrictions in genetics as to number of replications and possibilities to obtain offspring from combinations of parents.

Denote the model inputs by $x = (x_1 \ldots x_k)$, and denote the output studied by y. With the deterministic models studied here y is a function, f(x) say, of x. The variance of $y = f(x)$, induced by the distribution D of $x = (x_1 \ldots x_k)$ will be called VTOT

$$\text{VTOT} = \text{Var}[y] \qquad y = f(x), \quad x \sim D$$

Let S denote a subset of the x's, possibly one single x. The uncertainty contribution of subset S will be expressed in two ways. Firstly by the *top marginal variance*: the variance reduction that would occur in case one would get perfect new information about the inputs S. And secondly by the *bottom marginal variance*: the variance that will remain as long as one gets no new information about S. In both cases the new information is added to the information already present in input distribution D.

More formally, the variance that would remain in case input group S should become perfectly known, has the expectation E[Var[f(x)|S]]. Accordingly, the top marginal variance TMV_S of S is defined as

$$TMV_S = VTOT - E[Var[f(x)|S]].$$

Let ¬S indicate the complementary subset of all inputs not comprised in S. The variance that would remain in case ¬S should become perfectly known, has the expectation E[Var[f(x)|¬S]]. Thus we define the bottom marginal variance of S as

$$BMV_S = E[Var[f(x)|¬S]].$$

Obviously

$$BMV_S + TMV_{¬S} = VTOT.$$

The following well-known variance decomposition rule for conditional distributions

$$Var[y] = Var[E[y|S]] + E[Var[y|S]]$$

leads to an equivalent expression for TMV_S:

$$TMV_S = Var[E[y|S]].$$

For an independent group S,

$$BMV_S \geq TMV_S,$$

the possible difference being caused by interaction. See the literature cited for details like this.

In ASSA, top and bottom marginal variances are usually expressed as *fractions* of VTOT, and are then called *relative top and bottom marginal variances*.

When S consists of a single input x_i, $\eta^2_i \equiv TMV_i / VTOT$ is equal to the so-called *correlation ratio* of y and x_i. Note that the correlation ratio is not the same as the correlation coefficient. Only when $E[y|x_i]$ is linear in x_i, η^2_i is equal to the squared correlation coefficient between y and x_i, say ρ^2_i, but when $E[y|x_i]$ is nonlinear in x_i, η^2_i is greater than ρ^2_i.

When S consists of more than one component, $TMV_S / VTOT$ is also called the *correlation ratio, CR_S* say. The concept is not based upon a specific form of E[y|S] as function of S and it applies to a distribution rather than to a finite sample.

The relative bottom marginal variance can also be called the *complementary correlation ratio*, CCR_S say. From the mentioned relation $BMV_S + TMV_{¬S} = VTOT$, with ¬S denoting the complement of S, it follows that $CCR_S = 1 - CR_{¬S}$.

The next table mentions various names used in the literature for $TMV_S / VTOT$ and $BMV_S / VTOT$ which have the same meaning for an *independent* group S of inputs (of course the group S may also consist of a single input).

TMV_S / VTOT	BMV_S / VTOT
relative top marginal variance correlation ratio CR_S first order sensitivity index	relative bottom marginal variance complementary correlation ratio CCR_S total effect sensitivity index

Note

In general, TMV is a much more useful concept than BMV, and we advise to use BMV only in exceptional cases. TMV_S assesses the maximal improvement of prediction precision that might be attained by better knowledge about group S, or by better control of that group. If TMV_S is large, additional research about S might prove fruitful. If it would be utterly unrealistic, however, to expect to gain better knowledge about some input group S, you might use BMV_S to assess the uncertainty that would always remain even if you succeeded in eliminating all uncertainties about the other inputs. If, in such a case, BMV_S would be very large, research about the other inputs would seem rather futile.

10 Test functions

Simple functions, whose sensitivity-properties can be calculated analytically, are useful as examples to illustrate theoretical concepts. Moreover, they can be used to evaluate the performance of algorithms for SSA. And finally, they can be handy during software development as stand-in for true models. For the time being, only two such functions are mentioned. See Saltelli *et al.*, 2000 for many others.

10.1 Test function $sobol_8$

Non-monotonic test function $sobol_8$ (e.g. Saltelli *et al.*, 2000; Section 2.9) is defined as

$$sobol_8(x) = \Pi_{i=1\ldots8}\, g_i(x_i),$$

with

$$g_i(x_i) = (|4x_i - 2| + a_i) / (1+a_i) .$$

The x's are assumed to be independent homogeneous (0,1); and 8-vector a is given by

a = (0, 1, 4.5, 9, 99, 99, 99, 99).

The ASSA prototype of this function is

```
double sobol8(double *x)
```

where $x_{1\ldots8}$ is an existing 8-vector.

It is easy to see that the expectation of g_i is $E[g_i] = 1$, while $E[g_i^2] = 1 + \frac{1}{3}(1+a_i)^2$. By the independence of the g's, it follows that $E[sobol_8] = 1$, and that $VTOT = Var[sobol_8] = \Pi (E[g_i^2]) - 1$. The top marginal variances of the x_i's are given by $TMV_i = Var[g_i] / VTOT = (E[g_i^2] - 1) / VTOT$, while the bottom marginal variances of the x_i's equal $BMV_i = 1 - Var[\Pi_{j\neq i}(g_j)] / VTOT = 1 - \{\Pi_{j\neq i}(E[g_j^2]) - 1\} / VTOT$. The values of these variance components for the given a-parameters are presented in the next table.

Input	TMV/VTOT	BMV/VTOT
x_1	0.716192	0.787144
x_2	0.179048	0.242198
x_3	0.023676	0.034317
x_4	0.007162	0.010460
$x_{5\ldots8}$	0.000072	0.000105
Variance components of Sobol's test function, relative to VTOT=0.465424		

Note the difference in the last decimals with the values in the reference mentioned: possibly the latter values have been obtained by Monte Carlo methods. In this example, with independent inputs, and with many interactions, BMV > TMV, solely due to interactions.

10.2 Test function $normal_8$

Test function $normal_8$ is defined as

$$normal_8(x) = x_1^2 / \sqrt{2} + (x_2+x_3) / \sqrt{(7/4)} + 2\,(x_4-x_5) + x_6x_7\,\sqrt{2} + x_8.$$

The arguments are assumed to have the following distribution: the marginal distribution of each x_i is normal with mean 0 and variance 1; all correlations are equal 0, except $\rho(x_2,x_3)$ and $\rho(x_4,x_5)$, which equal 3/4.

The ASSA prototype of this function is

```
double normal8(double *x)
```

where $x_{1...8}$ is an existing 8-vector.

The function $normal_8(x)$ may be written as a *sum* of functions of independent groups, namely the groups $\{x_1\}$, $\{x_2,x_3\}$, $\{x_4,x_5\}$, $\{x_6,x_7\}$, $\{x_8\}$. They may be described as *independent groups with additive effects*. For such groups, the top and bottom marginal uncertainty contributions are equal. Thus one might speak unequivocally about *the* uncertainty contributions of these groups. They are listed in the following table.

Group	Absolute	Relative (%)
x_1	1	12.5
x_2, x_3	2	25
x_4, x_5	2	25
x_6, x_7	2	25
x_8	1	12.5
The uncertainty contributions of independent groups with additive effects.		

The next table gives the relative top and bottom marginal variances of the individual x's as percentage of the total variance. Note the large differences in the top and bottom marginal contributions of $x_2...x_6$. When x_2 is known, x_3 adds little information and vice versa. One might say that x_2 and x_3 are *exchangeable* in their effect on f. The opposite happens with x_4 and x_5: one might say that they are *complementary* in their effect on f. x_6 and x_7 are also complementary, be it in a different way, namely by their interaction.

Input	BMV	TMV
x_1	12.5	12.5
x_2	3.1	21.9
x_3	3.1	21.9
x_4	21.9	3.1
x_5	21.9	3.1
x_6	25	0
x_7	25	0
x_8	12.5	12.5
Top and bottom marginal variances (in %)		

11 Regression-based sensitivity analysis

11.1 General

For a long period, the most common form of SSA was an analysis via linear regression. In this chapter we describe the procedure: it performs an approximate SSA based on a linear regression approximation of the relation between the model inputs $x=(x_1...x_k)$ and the studied model output $y=f(x)$. The analysis starts with a sample of N draws from the uncertainty distribution of model input vector x. For each of these sampled x's the model output y has been calculated with the model software. The procedure, called `rsens_lin` Works by fitting various linear regressions to the sample. It keeps record of the residual mean squares of these linear models. The advantage of using `rsens_lin`, instead of fitting these models yourself, is that `rsens_lin` does the bookkeeping for you.

For instance, let a model output sample y have been calculated for an input sample of N vectors x. The correlation ratios, that is marginal variances, of grouped or individual x's can then be estimated by the procedure, which starts as follows.

```
void rsens_lin(long ngroup, long k, long N,
            double **X, double *y, long *key,
            double *rsquadlin, double *corrat, double *c_corrat)

/*
input ngroup: integer >= 2; number of input groups distinguished
input k: integer >= ngroup; number of model input variables
input N: integer > k+1; number of model runs
input X: k-by-N real matrix, rows contain model input variables
input y: real N-vector, the model output for varying inputs
input key: integer k-vector; grouping of the k model inputs
output rsquadlin: rsquared adjusted of the regression of y on all x-s
output corrat: real ngroup-vector; correlation ratio of the groups
output c_corrat: real ngroup-vector;complementary correlation ratios
*/
```

The results `corrat` and `c_corrat` are calculated as differences of *percentages of variance accounted for* (adjusted R^2). In the output of `rsens_lin`, negative values of `corrat` and `c_corrat` are replaced by 0.

The result `corrat` will contain the estimated correlation ratio of the groups distinguished, that is the relative top marginal variance. Result `c_corrat` will contain the estimated complementary correlation ratio, that is the relative bottom marginal variance.

In classical linear least squares regression, the y-variable is modelled as

$$y = \alpha_0 + \Sigma_{i\in I}\, \alpha_i x_i + \varepsilon$$

where I is some subset of {1...k} with unknown parameters α_i, and random ε. In our case, however, y is a deterministic function of more or less random x's, while the error is caused by the fact that f(x) cannot be written as $\alpha_0 + \Sigma\, \alpha_i x_i$. If the classical regression model applies, the residual mean square is an unbiased estimate of the expected squared error; but in our case the residual mean square will in general be biased as estimate of the expected squared error, even in the case of an ordinary random sample of x's. Fortunately, the bias shrinks to 0 for large sample sizes.

In the current version of ASSA, Rsquared-adjusted of the regression of y on a k-by-N matrix X is returned by the function

```
double rsquad(double *y, double **X, long k, long N)
```

Rsquared-adjusted is defined in terms of the mean-square, msy, of y (i.e. y's variance) and the mean-square, msr, of the regression residual (i.e. the residual variance).
These mean squares are derived from ssy, the sum-of-squares of y, and ssr, the sum of squares of the residual.

$R^2_{adjusted}$ = 1 - msr/msy = 1 - [ssr/(N-k-1)] / [ssy/(N-1)],

where ssr and ssy denote the total and residual sum of squares.

Function `rsquad` does not perform linear regression but merely calculates the regression residual via projection on the space orthogonal to the x-vectors (the rows of X) and the constant vector. The projection is done by Gramm-Schmidt orthogonalisation of the x-vectors. For this orthogonalisation `rsquad` internally uses an auxiliary procedure

```
void rop(double *res, double *y, double *x, long N)
```

which puts the residual of orthogonal projection of N-vector y on N-vector x into N-vector res.

Note

Instead of $R^2_{adjusted}$, some algorithms for regression-based SA use the multiple correlation coefficient, that is (unadjusted) R^2 = 1 - ssr/ssy. The difference is negligible if the number, k, of regressors is small compared with the sample size, N (as it should be). The adjustment for the number of regressors is mainly useful in the standard situation where the regression residuals are independent random, with equal variance. In the context of sensitivity analysis, there are no strong arguments for one type of R^2 in favour the other.

Example

```
#include "assa.h"
int main(int argc, char *argv[])
{
   long seed, N=1000, k=8, ngroup=5, *key, i, j;
   double **V, **X, *mu, *y, *x, rsquad, *corrat, *c_corrat;
   if (argc!=2) error("usage: xmprsens seed");
   seed = atol(argv[1]);
   if (seed > 0) seed = -seed;
   x = vector(1, k);
   mu = vector(1, k);
   key = ivector(1, k);
   corrat = vector(1, k);
   c_corrat = vector(1, k);
   y = vector(1, N);
   V = matrix(1, k, 1, k);
   X = matrix(1, k, 1, N);
   /* construct mean and variance matrix of multinormal inputs */
```

```
   for (i=1; i<=k; i++) {
      mu[i] = 0;
      for (j=1; j<=k; j++) V[i][j] = (i==j)? 1: 0;
   }
   V[3][2] = 0.75; V[5][4] = 0.75;
   /* take size-N ordinary random sample */
   mnor_mat(X, mu, V, k, N, &seed);
   /* calculate values y of testfunction normal8 */
   for (i=1; i<=N; i++) {
      for (j=1; j<=k; j++) x[j] = X[j][i];
      y[i] = normal8(x);
   }
   /* define the studied independent groups of inputs */
   key[1]=1; key[2]=key[3]=2; key[4]=key[5]=3;
      key[6]=key[7]=4; key[8]=5;
   printf("\nvariable: ");
   for (i=1; i<= k; i++) printf("%ld ", i);
   printf("\ngroup   : ");
   for (i=1; i<= k; i++) printf("%ld ", key[i]);
   printf("\n");

   /* linear-regression-based sensitivity analysis */
   rsens_lin(ngroup, k, N, X, y, key, &rsquad, corrat, c_corrat);
   printf("rsquad = %5.3f\n\n", rsquad);
   for (i=1; i<=ngroup; i++)
      printf("group%ld: corrat = %5.1f; c_corrat = %5.1f \n",
                  i, 100*corrat[i], 100*c_corrat[i]);
   /* here you can calculate bootstrap bootstrap interval */
   return 1;
}
#include "assa.c"
```

The example program is written in such a way that you can easily repeat the analysis, with different seeds, to get an impression of the sampling variability of the estimates. The next subsection takes another approach to sampling variability.

With `seed` 23032005 (actually you shouldn't take such systematic seeds), the program gives the following output.

```
variable: 1 2 3 4 5 6 7 8
group   : 1 2 2 3 3 4 4 5
rsquad = 0.645

group1: corrat =   0.0; c_corrat =   0.1
group2: corrat =  24.9; c_corrat =  22.9
group3: corrat =  30.8; c_corrat =  26.9
group4: corrat =   0.0; c_corrat =   0.2
group5: corrat =  11.1; c_corrat =  12.1
```

Note the low value 0.645 of R-squared-adjusted: only 65% of the variation of y can be explained by linear regression on the x's. From the section on test functions, we know that for each group the true values of the correlation ratio, CR, and the complementary correlation ratio, CCR, are equal in this example; the estimates, however, are seen to be different. The estimated CR and CCR of group 1 (x_1) are near zero, whereas we know that their true value is 12.5%. The same applies to group 4 (x_6 and x_7) which has 25% as theoretical CR and CCR. Nonlinear effects, here amounting to 37.5% of the variation, are simply not seen by this linear analysis. At the end of a linear regression-based sensitivity analysis, you always have to check if the unexplained variance, here

estimated as 35.5%, is small compared with the variance effects in which you are interested. This is not the case here, so we conclude that the analysis just performed has no practical value. In the next chapter we discuss a way-out.

11.2 Bootstrap percentile confidence interval

Since the N samples in the above example, were drawn *independently*, you can calculate a bootstrap confidence interval for the estimated sensitivity indices. The computational effort is negligible, since no additional model runs are required. For simplicity, we will use the percentile method as described in Efron and Tibshirani (1993, Chapter 13). The idea is that the sample's distribution is a fair approximation of the uncertainty distribution of the inputs, so that you may study properties of the uncertainty distribution by studying the sample distribution. From the N sampled input-vectors, and the corresponding outputs, a 'new' sample of size N is drawn several times, say `nboot` times. Procedure `boot`, draws such a new sample: it just draws N times, *with replacement*, from the numbers 1...N. Some of these numbers will occur more than once in the new sample, while others may be absent. This new sample, together with corresponding model output y, is analysed just as the original sample, yielding new sensitivity estimates, one for each fake sample, `nboot` all together. The α and $(1-\alpha)$ percentiles of these `nboot` estimates, constitute the bootstrap $(1-2\alpha)$ percentile confidence intervals.

The next C-block can be inserted in the above program, at the place indicated. It shows how to calculate a $1-2\alpha$ bootstrap percentile confidence interval for the 2-nd correlation ratio (of the 2-nd group: x_2 and x_3). Matrix `BX` is a bootstrap sample from the columns of original sample matrix `X`; vector `by` is filled with the N corresponding model outputs. Function `calc_quant` calculates the α and $(1-\alpha)$ quantiles of the nboot-vector `bc2`.

Example (continued)

```
{
   /* this block calculates a bootstrap percentile confidence
      interval: correlation ratio of group 2 as an example */
   long nboot=100, t, *bsam;
   double alpha=0.05, low, hig, estc2, **BX, *by, *bc2;
   bsam = ivector(1, N);
   BX = matrix(1, k, 1, N);
   by = vector(1, N);
   bc2 = vector(1, nboot);
   estc2 = corrat[2];
   printf("\nlinear regression based estimate of c2: %5.1f\n",
             100*estc2);
   for (t=1; t<=nboot; t++){
      boot(bsam, N, &seed);
      for (i=1; i<=N; i++) {
         by[i] = y[bsam[i]];
         for (j=1; j<=k; j++) BX[j][i] = X[j][bsam[i]];
      }
      rsens_lin(ngroup, k, N, BX, by, key,
                &rsquad, corrat, c_corrat);
      bc2[t] = corrat[2];
   }
   low = quantile(alpha, bc2, nboot);
```

```
      hig = quantile(1-alpha, bc2, nboot);
      printf("90 percent bootstrap confidence limits: ");
      printf("%5.1f  %5.1f\n", 100*low, 100*hig);
   }
```

With `seed` 23032005, inclusion of the above bootstrap block in the program gives the following additional output.

```
linear regression based estimate of c2:  24.9
90 percent bootstrap confidence limits:  21.0   28.7
```

12 Regression-free sensitivity analysis

We will illustrate the working of regression-free sensitivity analysis by means of an archetypical example. In a sense all other regression-free methods derive from this example.

To estimate the TMV of a stochastically *independent* subset S of the inputs, we construct two ordinary random samples of some size N, say X_1 and X_2, from the input distribution that have the *same* values for the inputs of S, while the other inputs are drawn independently. Next the two N-vectors of the studied model output, y_1 and y_2 say, are calculated, and from then on the values of the inputs can be forgotten, but the way the outputs were constructed still matters. The two sample variances v_1 and v_2 of the y-vectors are estimates of the total output variance VTOT. The geometric mean of these two is a combined estimate of VTOT.

$$\text{VTOT}\hat{} = \sqrt{(v_1\, v_2)}\ .$$

It is intuitively clear that the vectors y_1 and y_2 will be much the same if S is a very sensitive input group, and conversely. And indeed, it can be shown that the (Pearson) correlation between the vectors y_1 and y_2 is an estimate of the relative TMV, that is the correlation ratio.

$$(\text{TMV}_S / \text{VTOT})\hat{} = \text{corr}(y_1, y_2).$$

If you wish to estimate the BMV of the same subset S, calculate the following

$$(\text{BMV}_S / \text{VTOT})\hat{} = \text{mean}\,[\tfrac{1}{2}\,(y_{1i} - y_{2i})^2]\ /\ \sqrt{(v_1\, v_2)}\ .$$

(e.g. Sobol, 1990; Jansen, Rossing and Daamen, 1994; Saltelli *et al.*, 2000).

Example

In the next example we estimate the relative TMV, in %, of input group $S = \{x_2, x_3\}$ for the normal_8 test function. You may compare the result with the theory of Section 10, and the regression-based estimation of Section 11.

```
#include "assa.h"
int main(int argc, char *argv[])
{
   long N=1000, k=8, seed, i, j;
   double **V, **X1, **X2, *mu, *y1, *y2, *x,
          v1, v2, m1, m2, cov12, VTOT, TMV;
   if (argc!=2) error("usage: xmpanova seed");
   seed = atol(argv[1]);
   if (seed > 0) seed = -seed;
   x  = vector(1, k);
   mu = vector(1, k);
   y1 = vector(1, N); y2 = vector(1, N);
   V  = matrix(1, k, 1, k);
   X1 = matrix(1, k, 1, N); X2 = matrix(1, k, 1, N);

  /* construct mean-vector and variance matrix of multinormal inputs */
   for (i=1; i<=k; i++) {
```

```
      mu[i] = 0;
      for (j=1; j<=k; j++) V[i][j] = (i==j)? 1: 0;
   }
   V[3][2] = 0.75; V[5][4] = 0.75;
  /* take independent size-N ordinary random input samples X1 and X2 */
   mnor_mat(X1, mu, V, k, N, &seed);
   mnor_mat(X2, mu, V, k, N, &seed);
   /* equate the 2-nd and 3-rd rows of sample X2 to those of X1
      in order to calculate the TMV of the independent group
      formed by the 2-nd and 3-rd inputs */
   X2[2]=X1[2]; X2[3]=X1[3];
   /* calculate values y1 and y2 of test function normal8 */
   for (i=1; i<=N; i++) {
      for (j=1; j<=k; j++) x[j] = X1[j][i];
      y1[i] = normal8(x);
      for (j=1; j<=k; j++) x[j] = X2[j][i];
      y2[i] = normal8(x);
   }
   /* if you wish, you can delete the X-matrices: */
   free_matrix(X1, 1, k, 1, N); free_matrix(X2, 1, k, 1, N);
   /* calculate top marginal variance of x[2] and x[3] */
   m1=0; m2=0; v1=0; v2=0; cov12=0;
   for (i=1; i<=N; i++) {m1+=y1[i]; m2+=y2[i];}
   m1 /= N; m2 /= N;
   for (i=1; i<=N; i++)
      {v1+=SQR(y1[i]-m1); v2+=SQR(y2[i]-m2);
       cov12+=(y1[i]-m1)*(y2[i]-m2);}
   v1 /= (N-1); v2 /= (N-1); cov12 /= (N-1);
   VTOT = sqrt(v1*v2);
   TMV = 100 * cov12/VTOT;
   printf("tmv = %5.2f; vtot = %5.2f\n", TMV, VTOT);
   /* here you can calculate bootstrap confidence interval */
   return 1;
}
#include "assa.c"
```

With `seed` 1234567890, the analysis gives the output

```
tmv = 23.68; vtot =  7.66
```

Bootstrap percentile confidence interval

```
   {
      /* block calculating bootstrap percentile confidence interval
         for TMV of group with x2 and x3 */
      long nboot=100, *bsam;
      double alpha=0.05, low, hig, esttmv, *btmv, *by1, *by2;
      bsam = ivector(1, N);
      btmv = vector(1, nboot);
      by1 = vector(1, N); by2 = vector(1,N);
      esttmv = TMV;
      printf("\nregression-free estimate of TMV: %6.1f\n", esttmv);
      for (j=1; j<=nboot; j++){
         boot(bsam, N, &seed);
         for (i=1; i<=N; i++)
            {by1[i]=y1[bsam[i]]; by2[i]=y2[bsam[i]];}
         m1=0; m2=0;
         for (i=1; i<=N; i++) {m1+=by1[i]; m2+=by2[i];}
         m1 /= N; m2 /= N;
         v1=0; v2=0; cov12=0;
         for (i=1; i<=N; i++) {
```

```
            v1+=SQR(by1[i]-m1); v2+=SQR(by2[i]-m2);
            cov12+=(by1[i]-m1)*(by2[i]-m2);
         }
         v1 /= (N-1); v2 /= (N-1); cov12 /= (N-1);
         VTOT = sqrt(v1*v2);
         TMV = 100 * cov12/VTOT;
         btmv[j] = TMV;
      }
      low = quantile(alpha, btmv, nboot);
      hig = quantile(1-alpha, btmv, nboot);
      printf("90 percent bootstrap confidence limits: ");
      printf("%6.1f  %6.1f\n", low, hig);
   }
```

Inclusion of the above block in the main program, and running it with the same seed, yields this additional output

```
regression-free estimate of TMV:   23.7
90 percent bootstrap confidence limits:   18.5    28.0
```

References

Abramowitz, M. and Stegun, I.A., 1965, *Handbook of mathematical functions*, Dover, New York.

Efron, B. and Tibshirani, R.J., 1993, *An introduction to the bootstrap*, Chapman & Hall, London.

Comstock, R.E. and Robinson, H.F., 1948, *The components of genetic variance in populations of biparental progenies and their use in estimating the average degree of dominance*. Biometrics 4, 254-266.

Dey, Aloke, 1985, *Orthogonal fractional factorial designs,* Wiley Eastern Limited, New Delhi.

Fisher Box, J., 1978, *R.A. Fisher: the life of a scientist,* Wiley & Sons, New York.

Helton, J.C. and Davis, F.J., 2002, Latin hypercube sampling and the propagation of uncertainty in analyses of complex systems, Sandia National Laboratories, Albuquerque, Report SAND2001-0417.

Iman, R.L. and Conover, W.J., 1980, *Small sample sensitivity analysis techniques for computer models with an application to risk assessment*, Communications in Statistics A: theory and methods, 9, 1749-1842.

Iman, R.L. and Conover, W.J., 1982, *A distribution-free approach to inducing rank correlation among input variables*, Commun. statist.-simula. computa.,11(3), 311-334.

Jansen, M.J.W., Rossing, W.A.H. and Daamen, R.A., 1994, *Monte Carlo estimation of uncertainty contributions from several independent multivariate sources*, In: Grasman, J. and Van Straten, G. (eds.), *Predictability and Nonlinear Modelling in Natural Sciences and Economics*, p334-343, Kluwer, Dordrecht.

Jansen, M.J.W., 1996, *Winding stairs sample analysis program WINDINGS 2.0*, GLW-note MJA-96-2, GLW-DLO, Wageningen.

Jansen, M.J.W., 1997, *Maximum entropy distributions with prescribed marginals and normal score correlations*, Pages 87-92 in: Beneš V. & Štěpán (Eds.), Distributions with given marginals and moment problems, Kluwer, Dordrecht.

Jansen, M.J.W., 1999, *Analysis of variance designs for model output*, Computer Physics Communications 117, 35-43.

Kernighan and Ritchie, 1988, *The C programming Language*, second edition, Prentice Hall, New Jersey

McKay, M.D., Beckman, R.J. and Conover, W.J., 1979, *A comparison of three methods for selecting values of input variables in the analysis of output from a computer code*, Technometrics, 21, 239-245.

McKay, M.D., 1996, *Variance-based methods for assessing uncertainty importance in NUREG-1150 analyses*, Los Alamos National Laboratory, LA-UR-96-2695. Also available as http://www.jrc.it/uasa/services/samo_group/download/paper.ps

Owen, A.B., 1992, *A central limit theorem for latin hypercube sampling*, J.R.Statist.Soc.B 54, 541-551.

Press, W.H., Teukolsky, S.A., Vettering, W.T. and Flannery, B.P., 1992, *Numerical recipes in C: the art of scientific computing*, second edition, Cambridge University Press, Cambridge.

Press, W.H., Teukolsky, S.A., Vettering, W.T. and Flannery, B.P., 2002, *Numerical recipes in C++: the art of scientific computing*, second edition, Cambridge University Press, Cambridge.

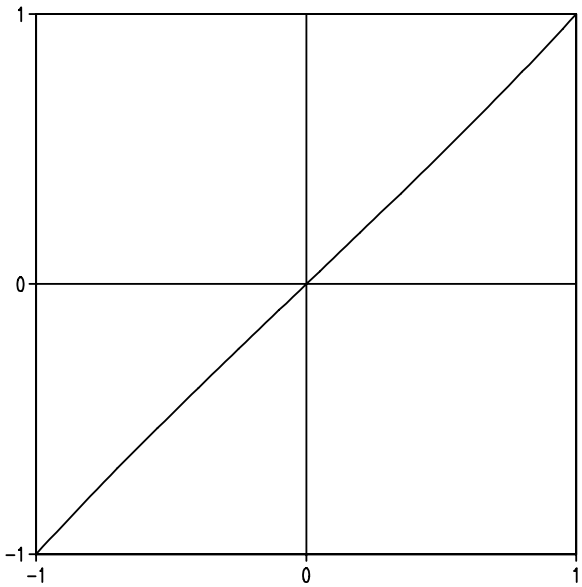
Saltelli, A., Tarantola, S. and Chan, K.P.S., 1999, *A quantitative model-independent method for global sensitivity analysis of model output*, Technometrics 41, 39-56.
Saltelli, A., Chan, K. and Scott, E.M., 2000, *Sensitivity analysis*, Wiley, Chichester.
Sobol, I.M., 1990, *Sensitivity estimates for nonlinear mathematical models*, Matematicheskoe Modelirovanie 2, 112-118 (in Russian), translated in Mathematical Modelling and Computational Experiments, vol 1, 407-414.
Sobol, I.M., 1995, *Sensitivity analysis of nonlinear models using sensitivity indices*, International symposium SAMO95: theory and applications of sensitivity analysis of model output in computer simulation, September 1995, Belgirate, Italy.
Stein, M., 1987, *Large sample properties of simulations using latin hypercube sampling*, Technometrics, 29, 143-151.

Appendix 1 Mathematical details

Ordinary random samples

In this appendix it will be shown how `uhc_corr` draws a sample from a continuous multivariate distribution with standard homogeneous marginals and rank correlations very close to the desired rank correlation. We could not easily find the material of this appendix in the SSA literature.

The procedure is based on the property that the Pearson and rank correlations of a multinormal distribution are very nearly equal (see Figure). Applying this property, `uhc_corr` works as follows. Firstly, a multinormal sample of k variables, say z_1 ... z_k, is drawn with mean 0, and covariance matrix C. The standard normal marginals z_i are transformed into standard homogeneous x_i by means of the mapping $x_i = \Phi(z_i)$, where Φ denotes the standard normal distribution function.



Bivariate normal distribution: rank correlation versus ordinary correlation. The two can be seen to be very nearly equal.

Pearson and rank correlation of the multinormal distribution

Before we can demonstrate the property mentioned, we have to introduce the concept of rank correlation for random variables, since, originally, rank correlation is only defined for samples. The distributional rank correlation between two continuous random variables x_1 and x_2, with marginals F_1 and F_2 is defined as the correlation between the corresponding standard homogeneous variables $F_1(x_1)$ and $F_2(x_2)$. Some authors use the term *grade correlation* instead.

Obviously, the distributional rank correlation between two standard homogeneous variables is equal to their ordinary Pearson correlation. Moreover, the distributional rank correlation between two variables is invariant under monotonically increasing transformations per variable. There exists a close connection between distributional and sample rank correlation: the sample rank correlation of a large ordinary random sample from a pair of variables with distributional rank correlation ρ^*, will tend to ρ^*.

It will be shown that the distributional rank correlation between any two variables standard homogeneous variables x_i and x_j from which `uhc_corr` draws a sample is close to the desired value c_{ij}:

$$\begin{aligned} \text{rcorr}(x_i, x_j) &= \text{corr}(x_i, x_j) \\ &= (6/\pi)\arcsin(c_{ij}/2) \\ &= c_{ij} + \eta_{ij} \end{aligned}$$

in which the approximation error η_{ij} satisfies $|\eta_{ij}| \leq 0.018$.

We will give a proof for the first two variables x_1 and x_2. Amazingly, no such proof was given by Iman and Conover (1982), who proposed the method. Denote their desired rank correlation c_{12} by ρ, and denote their actual rank correlation by ρ^*. The variables z_1 and z_2 are bivariate normal with standard normal marginals and correlation ρ. Thus, $x_1=\Phi(z_1)$ and $x_2=\Phi(z_2)$ are standard homogeneous; so both have mean 1/2 and variance 1/12. Their correlation may be calculated via the introduction of two auxiliary standard normal variables, ε_1 and ε_2 that are independent of each other and of z_1 and z_2. By the definition of Φ, one has

$$x_i = \Phi(z_i) = P(\varepsilon_i<z_i) = P(\varepsilon_i-z_i<0) ,$$

so that the expectation $E[x_1x_2]$ satisfies

$$E[\Phi(z_1)\Phi(z_2)] = E[P(\varepsilon_1-z_1<0|z_1)\, P(\varepsilon_2-z_2<0|z_2)] = P(\varepsilon_1-z_1<0 \cap \varepsilon_2-z_2<0).$$

Now ε_1-z_1 and ε_2-z_2 have normal distributions with mean 0, variance 2, and correlation $\rho/2$. The probability that both are negative is given by

$$P(\varepsilon_1-z_1<0 \cap \varepsilon_2-z_2<0) = 1/4 + \arcsin(\rho/2) / (2\pi)$$

(see for instance Abramowitz and Stegun, 1964; formula 26.3.19). So that

$$E[x_1x_2] = E[\Phi(z_1)\Phi(z_2)] = 1/4 + \arcsin(\rho/2) / (2\pi).$$

The correlation between x_1 and x_2 follows as $\rho^* = (6/\pi)\arcsin(\rho/2)$; which concludes the first part of the proof. The closeness of ρ^* to ρ is ascertained numerically: $\max(|\rho-\rho^*|)$ appears to have the value 0.018 (see figure).

A different interpretation

In the previous section it was shown that `uhc_corr` draws a sample from a continuous multivariate distribution with standard homogeneous marginals and rank correlations very close to the desired rank correlation. Note that a distribution is not uniquely defined by its marginals and correlation matrix, so that there are more distributions satisfying the specifications.

Amazingly, the procedure can also be interpreted in a different way: `uhc_corr` draws from the maximum-entropy distribution with standard homogeneous marginals and normal-score correlation matrix C. This distribution is unique, and its property of maximal entropy is attractive in the context of uncertainty analysis: of all distributions satisfying the given constraints, the one with maximal entropy contains the least information. Adopting any other distribution would be tantamount to assuming that we know more than we actually do (Jansen, 1997).

Restricted random samples: imposing correlations on samples

Procedure `iman_rank_mat` is also based on the near-equality of rank and Pearson correlations in the multinormal distribution. This procedure, which produces a rank-matrix with correlations very close to a correlation matrix given by the user. It is based on Iman and Conover (1982) and follows the clear summary of that paper by Helton and Davis (2002) using van der Waerden scores.

First of all, `iman_rank_mat` can be used to introduce rank correlations in an ordinary random sample e.g. from `uhc_basic`. But this method for introducing rank correlation into a sample is most often applied to latin hypercube samples. This happens in `luhc_iman`, which internally calls `iman_rank_mat`, and forces these ranks on a simple uncorrelated latin hypercube sample.

Appendix 2 NRC procedures required in ASSA

Routines

`double gammp(double a, double x)`
Cumulative gamma distribution with shape parameter a and scale parameter 1

`double gammln(double xx)`
Logarithm of gamma function

`void gser(double *gamser, double a, double x, double *gln)`
Incomplete gamma function via series development; used by `gammp`

`void gcf(double *gammcf, double a, double x, double *gln)`
Incomplete gamma function via continued fraction; used by `gammp`

`double betai(double a, double b, double x)`
Cumulative beta distribution with parameters a and b

`double betacf(double a, double b, double x)`
Cumulative beta(a,b) distribution via continued fraction

`void indexx(long n, double arr[], long indx[])`
Sorting algorithm; integer n-vector `indx` contains the indices of the smallest, next smallest … largest elements of real n-vector `arr`

`double ran1(long *idum)`
Real uniform 0-1 random generator

`double gasdev(long *idum)`
Standard normal random generator (mean 0; variance 1)

Frequently used utilities

`vector` and `free_vector`
(elements changed to double precision by preprocessor)

`matrix` and `free_matrix`
(elements changed to double precision by preprocessor)

`ivector` and `free_ivector`
(elements changed to long precision by preprocessor)

`imatrix` and `free_imatrix`
(elements changed to long by preprocessor)

Appendix 3 ASSA-1.0 header-file

```
/*
  assa header file
  version 1.0, may 2005
  Michiel Jansen
  Biometris, Wageningen-UR
*/


#include <stdio.h>
#include <math.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>


/* INCLUDE NRC ANSI PROTOTYPES */
#define float double
#define unsigned
#define int long
#ifndef ANSI
  #define ANSI
#endif
#include "nrutil.h"
#include "nr.h"
#undef float
#undef unsigned
#undef int


/* DEFINITION OF TYPE BOOLEAN */
typedef enum {false, true} boolean;


/* SPECIAL CONSTANTS */
#define ASSA_MISSING -99999.
#define ASSA_EPS 3.0e-7


/* ELEMENTARY PROCEDURES */

/* summary statistics */
void summary(double *x, long N,
        double *mean, double *var,
        double *min, double *max, double *med);
void msummary(double **X, long k, long N,
        double *mean, double *var, double *min, double *max,
        double *med, double **V);
double quantile(double prob, double *x, long N);
void calc_varcov(double **X, double **V, long k, long N);
void calc_corr(double **C, double **V, long k);
void calc_rank(double *x, double *rank, long N);
```

```
/* matrix operations */
void cholesky(double **S, double **L, long k);
void lowinv(double **L, double **LINV, long k);
boolean  posdef(double **S, long k);

/* utilities */
void error(char text[]);
void warning(char text[]);


/* BASIC RANDOM GENERATORS */
long any(long N, long* seed);
void perm(long* x, long N, long* seed);
void boot(long *x, long N, long *seed);
void mnor_mat(double **X, double *mean, double **V,
            long k, long N, long *seed);


/* PROBABILITY DISTRIBUTIONS AND RELATED SPECIAL FUNCTIONS */
double pnormal(double x);
double invnormal(double p);
void q2m_normal(double *mean, double *variance,
                double p1, double p2, double q1, double q2);

void q2m_lognormal(double *mean, double *variance,
                double p1, double p2, double q1, double q2);
void m2p_lognormal(double *mu, double *sigma,
                   double mean, double variance);

double ptriang(double x, double low, double top, double hig);
double invtriang(double p, double low, double top, double hig);

double pgamma(double x, double a, double b);
double invgamma(double p, double a, double b);
void m2p_gamma(double *a, double *b, double mean, double
variance);

double pbeta(double x, double a, double b);
double invbeta (double p, double a, double b);
void m2p_beta(double *a, double *b, double mean, double variance);


/* GENERATORS IN THE UNIT HYPERCUBE */
void uhc_basic(double **U, long k, long N, long *seed);
void uhc_corr(double **U, double **C, long k, long N, long *seed);
void luhc_basic(double **U, long k, long N, long *seed);
void rank_mat(long **R, double **C,
            long k, long N, long *seed);
void luhc_corr(double **U, double **C, long k, long N, long
*seed);
void iman_rank_mat(long **R, double **C,
            long k, long N, long *seed);
void luhc_iman(double **U, double **C, long k, long N, long
*seed);
void uhc_sama(double **U, long prime, long power, long *seed);
```

```
/* TEST FUNCTIONS */
double sobol8(double *x);
double normal8(double *x);


/* REGRESSION-BASED SENSITIVITY ANALYSIS */
double rsquad(double *y, double **X, long k, long N);
void rsens_lin(long ngroup, long k, long N,
            double **X, double *y, long *key,
            double *rsquadlin, double *corrat, double *c_corrat);
void rop(double *res, double *y, double *x, long N);
```

Appendix 4 ASSA-1.0 source-file

```
/*
  assa c-file
  version 1.0, may 2005
  Michiel Jansen, Biometris, Wageningen-UR
*/


/* INCLUDE NRC ANSI ROUTINES */
#define float double
#define unsigned
#define int long
#ifndef ANSI
  #define ANSI
#endif
#include "nrutil.c"
#include "ran1.c"
#include "gasdev.c"
#include "betacf.c"
#include "betai.c"
#include "gammln.c"
#include "gammp.c"
#include "gcf.c"
#include "gser.c"
#include "indexx.c"
#undef float
#undef unsigned
#undef int



/* ELEMENTARY PROCEDURES */

/* summary statistics */

void summary(double *x, long N,
        double *mean, double *var,
        double *min, double *max, double *med)
{
   long i, *ind;
   double z, alpha, min_, max_, mean_, var_;
   if (N<=1) error("summary: sample size <= 1");
   mean_ = 0; var_ = 0; min_ = x[1]; max_ = x[1];
   for (i=1; i<=N; i++) {
      z      = x[i];
      min_   = (min_ < z)? min_ : z;
      max_   = (max_ > z)? max_ : z;
      mean_ += z;
   }
   mean_ /= N;
   *mean = mean_;
   *min  = min_;
```

```
   *max  = max_;
   for (i=1; i<=N; i++) {
      z      = x[i] - mean_;
      var_  += z*z;
   }
   var_  /= (N-1);
   *var  = var_;
   ind = ivector(1, N);
   indexx(N, x, ind);
   i = (long) floor(0.5*N + 0.5);
   alpha = 1 - (0.5*N + 0.5 - i);
   *med  = alpha*x[ind[i]] + (1-alpha)*x[ind[i+1]];
   free_ivector(ind, 1, N);
}


void msummary(double **X, long k, long N,
       double *mean, double *var, double *min, double *max,
        double *med,
        double **V)
/* med and V calculated only if med and V don't point to NULL */
{
   long i, j, j1, j2, *ind;
   double z, mean_, min_, max_, var_, alpha;
   if (N<=1) error("msummary: sample size <= 1");
   if (med != NULL) ind = lvector(1, N);
   for (j=1; j<=k; j++){
      mean_=0; var_=0; min_=X[j][1]; max_=min_;
      for (i=1; i<=N; i++) {
         z      = X[j][i];
         min_   = (z<min_)? z : min_;
         max_   = (z>max_)? z : max_;
         mean_ += z;
      }
      mean_  /= N;
      mean[j] = mean_;
      min[j]  = min_;
      max[j]  = max_;
      for (i=1; i<=N; i++) {
         z      = X[j][i] - mean[j];
         var_  += z*z;
      }
      var_  /= (N-1);
      var[j]  = var_;
      if (med != NULL){
         indexx(N, X[j], ind);
         i = (long) floor(0.5*N + 0.5);
         alpha = 1 - (0.5*N + 0.5 - i);
         med[j]  = alpha*X[j][ind[i]] + (1-alpha)*X[j][ind[i+1]];
      }
   }
   if (med != NULL) free_lvector(ind, 1, N);
   if (V != NULL)
      for (j1=1; j1<=k; j1++) for (j2=1; j2<=j1; j2++) {
         z = 0;
         for (i=1; i<=N; i++) z += (X[j1][i]-mean[j1]) *(X[j2][i]-mean[j2]);
```

```
         V[j1][j2] =  z / (N-1);
      }
}


double quantile(double prob, double *x, long N)
{
   long i, *ind;
   double theta, qua;
   if (N<=1) error("quantile: sample size <= 1");
   ind = ivector(1, N);
   indexx(N, x, ind);
   i = (long) floor(prob*N + 0.5);
   if (i<1||i+1>N) error("quantile: sample too small");
   theta = 1 - (prob*N + 0.5 - i);
   qua = theta*x[ind[i]] + (1-theta)*x[ind[i+1]];
   free_ivector(ind, 1, N);
   return qua;
}


void calc_varcov(double **X, double **V, long k, long N)
{
   long i, j, j1, j2;
   double s, *mean;
   if (N<=1) error("calc_varcov: sample size <= 1");
   mean = vector(1, k);
   for (j=1; j<=k; j++){
      s=0;
      for (i=1; i<=N; i++) s += X[j][i];
      mean[j] = s/N;
   }
   for (j1=1; j1<=k; j1++) for (j2=1; j2<=j1; j2++) {
      s = 0;
      for (i=1; i<=N; i++) s += (X[j1][i]-mean[j1]) * (X[j2][i]-
mean[j2]);
      V[j1][j2] =  s / (N-1);
   }
   free_vector(mean, 1, k);
}


void calc_corr(double **C, double **V, long k)
{
   double z;
   long j1, j2;
   for (j1=1; j1<=k; j1++) for (j2=1; j2<=j1; j2++) {
      z = V[j1][j1]*V[j2][j2];
      if (z != 0) C[j1][j2] = V[j1][j2] / sqrt(z);
         else C[j1][j2] = ASSA_MISSING;
   }
}


void calc_rank(double *x, double *rank, long N)
{
```

```
   long i, *ind;
   ind = ivector(1, N);
   indexx(N, x, ind);
   for (i=1; i<=N; i++) rank[ind[i]] = i;
   free_ivector(ind, 1, N);
}



/* matrix operations */

void cholesky(double **S, double **L, long k)
{
   long i, j, m;
   double x;
   for (j=1; j<=k; j++) for (i=j; i<=k; i++){
      x = S[i][j];
      for (m=1; m<=j-1; m++) x -= L[j][m] * L[i][m];
      if (i==j){
         if (x<=0) error("cholesky: matrix not strictly positive definite");
         L[i][i] = sqrt(x);

      }
      else L[i][j] = x / L[j][j];
   }
   for (j=2; j<=k; j++) for (i=1; i<j;  i++) L[i][j] = 0;
}


void lowinv(double **L, double **LINV, long k)
{
   long i,j,m;
   double s;
   for (i=1; i<=k; i++){
      for (j=1; j<i; j++) LINV[j][i] = 0;
      LINV[i][i] = 1 / L[i][i];
      for (j=i+1; j<=k; j++){
         s=0;
         for(m=i; m<j; m++) s -= L[j][m]*LINV[m][i];
         LINV[j][i] = s / L[j][j];
      }
   }
}


boolean posdef(double **S, long k)
{
   long i, j, m;
   double **L;
   double x;
   boolean pd = true;
   L = matrix(1, k, 1, k);
   for (j=1; j<=k; j++) for (i=j; i<=k; i++){
      x = S[i][j];
      for (m=1; m<=j-1; m++) x -= L[j][m] * L[i][m];
      if (i==j){
```

```
         if (x<=0) {pd = false; goto clear;}
         L[i][i] = sqrt(x);
      }
      else L[i][j] = x / L[j][j];
   }
   clear:
   free_matrix(L, 1, k, 1, k);
   return pd;
}



/* utilities */

void error(char text[])
{
   fprintf(stderr,"\n\nFatal error:\n");
   fprintf(stderr,"%s\n", text);
   fprintf(stderr,"Bye-bye\n");
   exit(1);
}


void warning(char text[])
{
   fprintf(stderr,"\n\nWarning:\n");
   fprintf(stderr,"%s\n", text);
}



/* BASIC RANDOM GENERATORS */



long any(long N, long *seed)
{
   double t;
   t = N * ran1(seed);
   if (t==0) return 1;
   else return ((long)ceil(t));
}



void perm(long *x, long N, long *seed){
   long i, j;
   long* z;
   z = ivector(1,N);
   for (i=1; i<=N; i++) z[i] = i;
   for (i=N; i>=1; i--) {
      j = any(i, seed);
```

```
      x[i] = z[j];
      z[j] = z[i];
   }
   free_ivector(z,1,N);
}



void boot(long *x, long N, long *seed){
   long i;
   for (i=1; i<=N; i++) x[i] = any(N, seed);
}


void mnor_mat(double **X, double *mean, double **V,
              long k, long N, long *seed)
{
   long i,r,c;
   double *z;
   double **chol;
   z = vector(1,k);
   chol = matrix(1,k,1,k);
   if (posdef(V, k)== false)
       error("mnor_mat: V not positive definite");
   cholesky(V, chol, k);
   for (i=1; i<=N; i++) {
      for (r=1; r<=k; r++){
         z[r] = gasdev(seed);
         X[r][i] = mean[r];
         for (c=1; c<=r; c++) X[r][i] += chol[r][c] * z[c];
      }
   }
   free_matrix(chol,1,k,1,k);
   free_vector(z,1,k);
}



/* PROBABILITY DISTRIBUTIONS AND RELATED SPECIAL FUNCTIONS */



double pnormal(double x)
{  /* Abramowitz and Stegun */
   static double b[5] =
   { 0.319381530, -0.356563782, 1.781477937, -1.821255978, 1.330274429 };
   static double p  = 0.2316419;
   static double pi = 3.1415927;
   double P, t;
   long j;
   if (x<-10.0) return pnormal(-10.0);
   if (x>10.0)  return pnormal(10.0);
   if (x>=0.0) t=1/(1+p*x); else t=1/(1-p*x);
   P = 0;
```

```
   for (j=4; j>=0; j--) P=(b[j]+P)*t;
   P = P*exp(-x*x/2.0)/sqrt(2.0*pi);
   if (x>=0) return 1-P; else return P;
}


double invnormal(double p)
{
   /* Abramowitz and Stegun */
   static double c[3]= {2.515517, 0.802853, 0.010328};
   static double d[4]= {1.0, 1.432788, 0.189269, 0.001308};
   double q, t, xp, numerator, denominator;
   if (p<=.0||p>=1.0)
      {warning("invnormal: invalid argument\n"); return ASSA_MISSING;}
   q= 1-p;
   if (q>.5) q=p;
   t= sqrt(-2.0*log(q));
   denominator = d[0]+t*(d[1]+t*(d[2]+t*d[3]));
   numerator = c[0]+t*(c[1]+t*c[2]);
   xp = t - numerator/denominator;
   if (p<.5) xp = -xp;
   return xp;
}



void q2m_normal(double *mean, double *variance,
                double p1, double p2, double q1, double q2)
{
   double e1, e2, a, b;
   if (p1<=0 || p2<=0 || p1>=1 || p2>=1)
      error("q2m_normal: p1 or p2 out of range");
   if (p1==p2 || q1==q2) error("q2m_normal: p1=p2 or q1=q2");
   if ((p1-p2)/(q1-q2) < 0) error("q2m_normal: incompatible p1,p2,q1,q2");
   e1 = invnormal(p1); e2=invnormal(p2);
   /* solve q[i] = a + b*e[i] */
   b = (q1-q2)/(e1-e2);
   a = q1 - b*e1;
   *mean = a;
   *variance = b*b;
}



void q2m_lognormal(double *mean, double *variance,
                double p1, double p2, double q1, double q2)
{
   double e1, e2, lq1, lq2, a, b, mu, sigmasq;
   if (p1<=0 || p2<=0 || p1>=1 || p2>=1)
      error("q2m_lognormal: p1 or p2 out of range");
   if (q1<=0 || q2<=0)
      error("q2m_lognormal: q1 or q2 <= 0");
   if (p1==p2 || q1==q2) error("q2m_lognormal: p1=p2 or q1=q2");
   if ((p1-p2)/(q1-q2) < 0) error("q2m_lognormal: incompatible p1,p2,q1,q2");
   e1 = invnormal(p1); e2=invnormal(p2);
```

```
   lq1 = log(q1); lq2 = log(q2);
   /* solve lq[i] = a + b*e[i] */
   b = (lq1-lq2)/(e1-e2);
   a = lq1 - b*e1;
   mu = a;
   sigmasq = b*b;
   *mean = exp(mu + sigmasq/2);
   *variance = (*mean)*(*mean) * (exp(sigmasq)-1);
}


void m2p_lognormal(double *mu, double *sigma,
                   double mean, double variance)
{
   if (mean<=0 || variance <= 0)
      error("m2p_lognormal: mean or variance out of range");
   *sigma = sqrt(log(variance/(mean*mean)+1));
   *mu = log(mean) - (*sigma)*(*sigma)/2;
}


double ptriang(double x, double low, double top, double hig)
{
   double xi, tau;
   if (low >= hig) error("ptriang: low >= hig");
   if (top<low || top > hig) error("ptriang: top not between low and hig");
   if (x<=low) return 0.;
   if (x>=hig) return 1.;
   xi = (x-low)/(hig-low);
   tau = (top-low)/(hig-low);
   return (xi<=tau)? xi*xi/tau : 1-(1-xi)*(1-xi)/(1-tau);
}


double invtriang(double p, double low, double top, double hig)
{
   double xi, tau;
   if (low >= hig) error("invtriang: low >= hig");
   if (top<low || top > hig) error("invtriang: top not between low and hig");
   if (p<0 || p>1) error("invtriang: p not between 0 and 1");
   if (p==0) return low;
   if (p==1) return hig;
   tau = (top-low)/(hig-low);
   xi = (p<=tau)? sqrt(p*tau) : 1-sqrt((1-p)*(1-tau));
   return low + xi*(hig-low);
}


double pgamma(double x, double a, double b)
{
   if (x<0.0 || a<=0 || b<=0)
   {warning("Pgamma: invalid arguments"); return ASSA_MISSING;}
   if (x==0) return 0;
   return gammp(a, x/b);
}
```

```
double invgamma(double p, double a, double b)
{
   /* initial bracketing followed by bisection: might be slow */
   double low = 0.0, mid, hig = 1.0;
   static double BIGA = 400, BIG = 1e+37;
   if (p<0.0||p>1.0||a<=0||b<=0)
      {warning("Invgamma: invalid arguments"); return ASSA_MISSING;}
   if (p==0) return 0; if (p==1) return BIG;
   if (a > BIGA) { /* Wilson-Hilferty approximation */
      double w;
      w = invnormal(p)/sqrt(9*a) + 1 - 1/(9*a);
      return b * a * w*w*w;
   }
   while (gammp(a, hig/b)<p) {low = hig; hig = 2*hig;}
   while (hig-low>ASSA_EPS){
      mid = (low+hig)/2;
      if (gammp(a, mid/b)>=p) hig = mid; else low = mid;
   }
   return hig;
}


void m2p_gamma(double *a, double *b, double mean, double variance)
{
   *a = mean*mean/variance;
   *b = variance/mean;
}


double pbeta(double x, double a, double b)
{
   if (x<0.0||x>1.0||a<=0||b<=0)
      {warning("Pbeta: invalid arguments"); return ASSA_MISSING;}
   if (x==0) return 0;
   if (x==1) return 1;
   return betai(a, b, x);
}


double invbeta(double p, double a, double b)
{
   double low = 0.0, mid, hig = 1.0;
   if (p<0.0||p>1.0||a<=0||b<=0)
      {warning("Invbeta: invalid arguments"); return ASSA_MISSING;}
   if (p==0.0) return 0.0; if (p==1.0) return 1.0;
   while (hig-low>ASSA_EPS){
      mid = (low+hig)/2;
      if (betai(a, b, mid)>=p) hig = mid; else low = mid;
   }
   return hig;
}


void m2p_beta(double *a, double *b, double mean, double variance)
{
```

```
   double z;
   if ((variance<=0.) || (variance>=mean*(1-mean)) || (mean<=0.) || (mean>=1.))
      error("m2p_beta: invalid mean and variance\n");
   z = variance/mean + mean;
   *a = (1-z) * mean / (z - mean);
   *b = (1-z) * (1-mean) / (z - mean);
}



/* GENERATORS IN THE UNIT HYPERCUBE */

void uhc_basic(double **U, long k, long N, long *seed)
{
   long i,j;
   for (i=1; i<=k; i++) for (j=1; j<=N; j++) U[i][j] = ran1(seed);
}


void uhc_corr(double **U, double **C, long k, long N, long *seed)
{
   long i,j;
   double *mean;
   if (posdef(C, k)== false)
      error("uhc_corr: cormat not positive definite");
   mean = vector(1,k);
   for (i=1; i<=k; i++) mean[i]=0.;
   mnor_mat(U, mean, C, k, N, seed);
   for (i=1; i<=k; i++) for (j=1; j<=N; j++)
         U[i][j] = pnormal(U[i][j]);
   free_vector(mean, 1, k);
}


void luhc_basic(double **U, long k, long N, long *seed)
{
   long i,j;
   long *z;
   z = ivector(1,N);
   for (i=1; i<=k; i++) {
      perm(z,N,seed);
      for (j=1; j<=N; j++) U[i][j] = (z[j] - 1 + ran1(seed))/N;
   }
   free_ivector(z,1,N);
}


void rank_mat(long **R, double **C,
              long k, long N, long *seed)
{
   /* matrix of sample ranks for given rankcorrelation */
   long i, j, r, *p;
   double *mean, **M;
   p = ivector(1,N);
   mean = vector(1, k);
```

```
   M = matrix(1,k,1,N);
   for (j=1; j<=k; j++)
      if (C[j][j] != 1.) error("rank_mat: diagonal c != 1");
   if (posdef(C,k) == false)
      error("rank_mat: C not positive definite");
   for (i=1; i<=k; i++) mean[i]=0.;
   mnor_mat(M, mean, C, k, N, seed);
   for (r=1; r<=k; r++){
      indexx(N, M[r], p);
      for (i=1; i<=N; i++) R[r][p[i]] = i;
   }
   free_matrix(M,1,k,1,N);
   free_vector(mean, 1, k);
   free_ivector(p,1,N);
}


void luhc_corr(double **U, double **C, long k, long N, long *seed)
{
   double **M;
   long i, j, *ind, **ranks;
   M = matrix(1,k,1,N);
   ranks = imatrix(1,k,1,N);
   ind = ivector(1,N);
   luhc_basic(M, k, N, seed);
   rank_mat(ranks, C, k, N, seed);
   for (i=1; i<=k; i++){
      indexx(N, M[i], ind);
      for (j=1; j<=N; j++) U[i][j] = M[i][ind[ranks[i][j]]];
   }
   free_ivector(ind,1,N);
   free_imatrix(ranks,1,k,1,N);
   free_matrix(M,1,k,1,N);
}


void iman_rank_mat(long **R, double **C,
                   long k, long N, long *seed)
{
   /* matrix of ranks for method of iman */
   long i, j, r, c, *p;
   double *z, *waerden, **sigma, **chol, **invchol, **M1, **M2;
   z = vector(1,k);
   sigma = matrix(1, k, 1, k);
   p = ivector(1,N);
   waerden = vector(1,N);
   chol = matrix(1,k,1,k);
   invchol = matrix(1,k,1,k);
   M1 = matrix(1,k,1,N);
   M2 = matrix(1,k,1,N);
   if (N<k) error("iman_rank_mat: sample size less than number of variates");
   for (j=1; j<=k; j++)
      if (C[j][j] != 1.) error("iman_rank_mat: diagonal C != 1");
   if (posdef(C,k) == false)
      error("iman_rank_mat: C not positive definite");
   for (i=1; i<=N; i++) waerden[i] = invnormal(i/(N+1.0));
```

```
   for (j=1; j<=k; j++){
      perm(p, N, seed);
      for (i=1; i<=N; i++) M1[j][i] = waerden[p[i]];
   }
   for (r=1; r<=k; r++) for (c=1; c<=r; c++){
      sigma[r][c] = 0;
      for (i=1; i<=N; i++) sigma[r][c] += M1[r][i]*M1[c][i];
      sigma[r][c] /= N;
   }
   if (!posdef(sigma,k)) error("iman_rank_mat: sample size too small?");
   cholesky(sigma, chol, k);
   lowinv(chol, invchol, k);
   /* make uncorrelated M2 */
   for (i=1; i<=N; i++) {
      for (r=1; r<=k; r++){
         M2[r][i] = 0;
         for (c=1; c<=r; c++) M2[r][i] += invchol[r][c] * M1[c][i];
      }
   }
   /* let M1 have the required correlations */
   cholesky(C, chol, k);
   for (i=1; i<=N; i++) {
      for (r=1; r<=k; r++){
         M1[r][i] = 0;
         for (c=1; c<=r; c++) M1[r][i] += chol[r][c] * M2[c][i];
      }
   }
   for (r=1; r<=k; r++){
      indexx(N, M1[r], p);
      for (i=1; i<=N; i++) R[r][p[i]] = i;
   }
   free_matrix(M2,1,k,1,N);
   free_matrix(M1,1,k,1,N);
   free_matrix(invchol,1,k,1,k);
   free_matrix(chol,1,k,1,k);
   free_vector(waerden,1,N);
   free_ivector(p,1,N);
   free_matrix(sigma, 1, k, 1, k);
   free_vector(z,1,k);
}


void luhc_iman(double **U, double **C, long k, long N, long *seed)
{
   double **M;
   long i, j, *ind, **ranks;
   M = matrix(1,k,1,N);
   ranks = imatrix(1,k,1,N);
   ind = ivector(1,N);
   luhc_basic(M, k, N, seed);
   iman_rank_mat(ranks, C, k, N, seed);
   for (i=1; i<=k; i++){
      indexx(N, M[i], ind);
      for (j=1; j<=N; j++) U[i][j] = M[i][ind[ranks[i][j]]];
   }
   free_ivector(ind,1,N);
```

```
   free_imatrix(ranks,1,k,1,N);
   free_matrix(M,1,k,1,N);

}


void uhc_sama(double **U, long prime, long power, long *seed)
{
     /*
     saturated main effect fractional factorial design
     (prime^power-1)/(prime-1) orthogonal factors at prime levels
     cf. Aloke Dey, 1985, Orthogonal fractional factorial designs, 2.4.1
     factors randomized, augmented with random term and scaled to unit interval
     still to do: allow less than maximum number of factors
     */
     long ncol; /* number of factors (columns) */
     long *x; /* x[1...power]: generating factors */
     long *k; /* k[1...power]: column specific coefficients of x[1...power] */
     long *z;
     long lead;   /* leading coefficient of sum(k[j]*x[j]) */
     long row, col, nrow, i, j, a, cnt;
     const long unset=-1;
     if (prime<2) error("uhc_sama: prime should be at least 2");
     if (power<2) error("uhc_sama: power should be at least 2");
     for (i=2; i<=prime-1; i++)
        if (prime%i==0) error("uhc_sama: value is not a prime");
     nrow = (long)pow(prime,power);
     ncol = (nrow-1) / (prime-1);
     x = ivector(1,power);
     k = ivector(1,power);
     z = ivector(1, prime);
     for (row=0; row<nrow; row++){
          a = row;
          for (i=power; i>=1; i--) {x[i] = a%prime; a /= prime;}
          cnt = 0;
          for (i=0; i<nrow; i++) {
               a=i; lead = unset;
               for (j=power; j>=1; j--) {
                    k[j] = a%prime;
                    if (lead==unset && k[j]!=0){
                         lead = k[j];
                         if (lead!=1) goto exit;
                    }
                    a /= prime;
               }
               if (lead==1) {
                    cnt++;
                    a = 0;
                    for (j=1; j<=power; j++) a += k[j]*x[j];
                    U[cnt][row+1] = a%prime;
               }
               exit:;
          }
     }
     for (col=1; col<=ncol; col++){
         perm(z, prime, seed);
```

```
         for (row=1; row<=nrow; row++)
            U[col][row] = (z[(long)floor(U[col][row]+
                                 1.001)]-1 + ran1(seed)) / prime;
      }
      free_ivector(z,1,prime);
      free_ivector(k,1,power);
      free_ivector(x,1,power);
}



/* TEST FUNCTIONS */

double sobol8(double *x)
{
   /* sobol test function, Saltelli et al. 2000 section 2.9
      x_1...x_8 assumed independent homogeneous on 0-1      */
   static double z[8] = {0,1,4.5,9,99,99,99,99};
   long i;
   double s=1;
   double *a;
   a = z-1;
   for (i=1; i<=8; i++) s *= (fabs(4*x[i]-2)+a[i]) / (1+a[i]);
   return s;
}



double normal8(double *x)
{
   /* test function for 8 standard normal x's
      zero correlation, except rho[3,2]=rho[5,4]=0.75  */
   return x[1]*x[1]/sqrt(2) + (x[2]+x[3])/sqrt(1.75) +
          (x[4]-x[5])*2 + x[6]*x[7]*sqrt(2) + x[8];
}



/* REGRESSION-BASED SENSITIVITY ANALYSIS */

double rsquad(double *y, double **X, long k, long N)
{
   /* calculates rsquared-adjusted of linear regression of y
      on the constant term and the rows of X */
   long i, j, j1, j2;
   double vary=0, rms=0, *res, *one, **X_;
   if (N-1-k <= 0) error("rsquad: N-k-1 <= 0");
   X_ = matrix(1, k, 1, N);
   one = vector(1, N);
   res = vector(1, N);
   for (i=1; i<=N; i++) one[i]=1;
   rop(res, y, one, N);
   for (i=1; i<=N; i++) vary += res[i]*res[i];
```

```
   vary /= (N-1);
   if (vary == 0) error("rsquad: y is constant");
   for (j=1; j<=k; j++) rop(X_[j], X[j], one, N);
   for (j1=1; j1<=k; j1++) {
      rop(res, res, X_[j1], N);
      if (j1<k) for (j2=j1+1; j2<=k; j2++) rop(X_[j2], X_[j2], X_[j1], N);
   }
   for (i=1; i<=N; i++) rms += res[i]*res[i];
   rms /= (N-1-k);
   free_vector(res, 1, N);
   free_vector(one, 1, N);
   free_matrix(X_, 1, k, 1, N);
   if (rms<vary) return 1 - rms/vary;
   return 0;
}


void rsens_lin(long ngroup, long k, long N,
               double **X, double *y, long *key,
               double *rsquadlin, double *corrat, double *c_corrat)
   /*
   input ngroup: integer >= 2; number of input groups distinguished
   input k: integer >= ngroup; number of model input variables
   input N: integer > k+1; number of model runs
   input X: k-by-N real matrix, rows contain model input variables
   input y: real N-vector, the model output for varying inputs
   input key: integer k-vector; grouping of the k model inputs
   output rsquadlin: rsquared adjusted of the regression of y on all x's
   output corrat: real ngroup-vector; correlation ratio of the groups
   output c_corrat: real ngroup-vector complementary correlation ratios
   */
{
   double **present, **absent;
   long  group, j, npres, nabs;
   boolean grouppresent;
   if (ngroup<2) error("rsens_lin: ngroup < 2");
   if (k<ngroup) error("rsens_lin: k < ngroup");
   if (N<=k+1) error("rsens_lin: N <= k+1");
   for (j=1; j<=k; j++) if (key[j]<1 || key[j] >ngroup)
      error("rsens_lin: out-of-bounds grouping in key");
   /* check if all levels 1...ngroup present in key */
   for (group=1; group<=ngroup; group++){
      grouppresent=false;
      for (j=1; j<=k; j++)
         if (key[j]==group) {grouppresent=true; break;}
      if (grouppresent==false)
         error("rsens_lin: incomplete key");
   }
   present = (double **) malloc((size_t)((k+1)*sizeof(double*)));
   absent  = (double **) malloc((size_t)((k+1)*sizeof(double*)));
   if (!present || !absent) error("rsens_lin: insufficient memory");
   *rsquadlin = rsquad(y, X, k, N);
   for (group=1; group<=ngroup; group++){
      npres=0; nabs=0;
      for (j=1; j<=k; j++){
         if (key[j]==group) {npres++; present[npres] = X[j];}
```

```
                    else {nabs++;  absent[nabs] = X[j];}
      }
      corrat[group] = rsquad(y, present, npres, N);
      c_corrat[group] = *rsquadlin - rsquad(y, absent, nabs, N);
   }
   free((char*) present);
   free((char*) absent);
}


void rop(double *res, double *y, double *x, long N)
{
   double xy=0, xx=0, alpha;
   long i;
   for (i=1; i<=N; i++){
      xy += x[i]*y[i];
      xx += x[i]*x[i];
   }
   alpha = (xx>0)? xy/xx : 0;
   for (i=1; i<=N; i++) res[i] = y[i]-alpha*x[i];
}
```

WOt-onderzoek

Verschenen werkdocumenten
in de reeks Wettelijke Onderzoekstaken – vanaf mei 2005

Werkdocumenten zijn verkrijgbaar bij het secretariaat van Wettelijke Onderzoekstaken Natuur & Milieu (voorheen Natuurplanbureau), gebouw Alterra-Oost, te Wageningen.

T 0317 – 47 78 45
F 0317 – 42 49 88
E info@npb-wageningen.nl

De werkdocumenten zijn ook te downloaden via de WOt-website
www.wotnatuurenmilieu.wur.nl

2005

1 *Eimers, J.W. (Samenstelling)*
Projectverslagen 2004.

2 *Hinssen, P.J.W.*
Strategisch Plan van de Unit Wettelijke Onderzoekstaken Natuur & Milieu, 2005 – 2009.

3 *Sollart, K.M.*
Recreatie: Kennis- en datavoorziening voor MNP-producten. Discussienotitie.

4 *Jansen, M.J.W.*
ASSA: Algorithms for Stochastic Sensitivity Analysis. Manual for version 1.0.

5 *Goossen, C.M. & S. de Vries*
Beschrijving recreatie-indicatoren voor de Monitoring en Evaluatie Agenda Vitaal Platteland (ME AVP)

6 *Mol-Dijkstra, J.P.*
Ontwikkeling en beheer van SMART2-SUMO. Ontwikkelings- en beheersplan en versiebeheerprotocol.

7 *Oenema, O.*
How to manage changes in rural areas in desired directions?

8 *Dijkstra, H.*
Monitoring en Evaluatie Agenda Vitaal Platteland; inventarisatie aanbod monitoringsystemen.

9 *Ottens, H.F.L & H.J.A.M. Staats*
BelevingsGIS (versie2). Auditverslag.

10 *Straalen, F.M. van*
Lijnvormige beplanting Groene Woud. Een studie naar het verdwijnen van lanen en perceelsrandbegroeiing in de Meierij

11 *Programma Commissie Natuur*
Onderbouwend Onderzoek voor de Natuurplanbureau-functie van het MNP; Thema's en onderzoeksvragen 2006