Using mistral for reliability analysis

Clément Walter

2017-04-11

The package `mistral` provides some numerical methods for estimating the probability that a computer code exceeds a given threshold when its input parameters are random with a known distribution. In this vignette we give an overview of the available methods as well as some practical testcases to clarify both how to use them and what they do.

General setting

Given an input random vector $\mathbf{X}$ with known distribution and a real-valued function g standing for the computer code, the general problem addressed in this package is to estimate the quantity:

$$p = P[g(\mathbf{X}) > q]$$

with p or q a given real:

- either q is given and the goal is to estimate the probability p; in this setting q is for instance a security threshold and one wants to estimate the probability that the code be greater than this value.
- or p is given and the goal is to find the corresponding threshold such that the probability of failure is lower than required.

The standard input space

By definition, we call the standard input space the case where $\mathbf{X}$ is a standard Gaussian vector, ie. a vector with independent standard Gaussian coordinates. **All the stochastic methods are developed for a standard Gaussian input space.** In other words, when the problem at hand does not use only independent standard Gaussian random variables, an iso-probabilistic transformation has to be done before a call to the code `g`, often called the limit-state function `lsf`.

`mistral` provides a way to perform such transformations for model using correlated inputs with usual distributions. In the following the original vector is denoted by $\mathbf{X}$ while its conterpart in the standard space is denoted by $\mathbf{U}$. The two functions `mistral::UtoX` and `mistral::XtoU` let go from one representation to the other.

Let us detail the use of `UtoX` (and similarly of `XtoU`). The supported distributions are:

- Normal: defined by its mean and standard deviation
- Lognormal: defined by its internal parameters `P1=meanlog` and `P2=sdlog` (see `*lnorm` help page)
- Uniform: defined by its internal parameters `P1=min` and `P2=max` (see `*unif` help page)
- Gumbel: defined by its internal parameters `P1` and `P2`
- Weibull: defined by its internal parameters `P1=shape` and `P2=scale` (see `*weibull` help page)
- Gamma: defined by its internal parameters `P1=shape` and `P2=scale` (see `*gamma` help page)
- Beta: defined by its internal parameters `P1=shape1` and `P2=shape2` (see `*beta` help page)

Let us define for instance a random vector $\mathbf{X}$ in 2d such that its coordinates are standard Gaussian:

```
distX1 <- list(type='Norm',  MEAN=0.0, STD=1.0, P1=NULL, P2=NULL, NAME='X1')
distX2 <- list(type='Norm',  MEAN=0.0, STD=1.0, P1=NULL, P2=NULL, NAME='X2')
input.margin <- list(distX1,distX2)
```

and correlated $cor(X_1, X_2) = 0.5$:

```
input.Rho    <- matrix( c(1.0, 0.5,
                          0.5, 1.0),nrow=2)
input.R0     <- mistral::ModifCorrMatrix(input.Rho)
L0           <- t(chol(input.R0))
```

then the function `UtoX` writes:

```
U <- rnorm(2)
U <- cbind(U, U)
X <- mistral::UtoX(U, input.margin, L0)
X
```

```
##            [,1]       [,2]
## [1,] -0.8573004 -0.8573004
## [2,] -1.5142190 -1.5142190
```

The function `UtoX` works with vectors or matrices and always returns a matrix. Eventually the limit-state function can be defined by:

```
lsf_U = function(U) {
    X <- mistral::UtoX(U, input.margin, L0)
    lsf(X)
}
```

Defining a proper limit-state function `lsf`

All the methods implemented in `mistral` and presented in this vignette require that `lsf` be a function taking a matrix as input and returning a vector as ouput. Indeed, in order to allow for parallel computing, batches of points to be evaluated are given as a matrix of column vectors. Hence, depending on the processing capaiblities and/or the implementation, computation of the model g on the points $\mathbf{X}$ can be made a lot faster.

Let us define an easy 2-dimensional function:

```
lsf <- function(x){
  x[1] + x[2]
}
```

This function can be called onto a vector:

```
x <- c(1,2)
lsf(x)
```

```
## [1] 3
```

or a matrix:

```
x <- as.matrix(x)
lsf(x)
```

```
## [1] 3
```

However, one wants it to be able to address matrices with several columns representing different points:

```
x <- cbind(x,x)
lsf(x)
```

```
## [1] 3
```

This obviously does not provide the expected result. There are indeed several possibilities to make the example fit into our framework. In this simple case, one has an analytical expression of the function. This

can arise when using the package for educational/illustrative purpose. An easy way to fix that is then to redefine the function with `x` as a matrix:

```
lsf <- function(x) {
  x[1,] + x[2,]
}
lsf(x)
```

```
## [1] 3 3
```

Note here that the function considers each **column** as a single point. This is to be consistent with the default behaviour of `as.matrix` function:

```
x <- 1:2
as.matrix(x)
```

```
##      [,1]
## [1,]    1
## [2,]    2
```

Now looking back at our function, one has:

```
x <- cbind(x,x)
lsf(x[,1])
```

```
## Error in x[1, ]: incorrect number of dimensions
```

The function returns an error because default R behaviour is to drop the dimensions when selecting only one column. All together, a robust way to define a function is to apply first `as.matrix`:

```
lsf <- function(x) {
  x <- as.matrix(x)
  x[1,] + x[2,]
}
lsf(x[,1])
```

```
## [1] 3
```

In general (practical) settings, no analytical expression is available. Let us denote by `myCode` the given computer code. `myCode` is supposed to be able to be called onto a vector. Then the `apply` function can be used:

```
lsf <- function(x){
  x <- as.matrix(x)
  apply(x, 2, myCode)
}
```

When parallel computing is availble, it is then possible to make a parallel calculation of a batch of points given as a matrix. Let us give an example using the `foreach` and `iterators` packages (we recommand the user to read their very nice vignettes to get started with the `foreach` loop, which is useful not only for parallel computing but also as a nice alternative to the `*apply` family).

```
require(foreach)
lsf <- function(x){
  x <- as.matrix(x)
  foreach(x = iterators::iter(x, by = 'col'), .combine = 'c') %dopar% {
    myCode(x)
  }
}
```

Giving known points in input

Some methods implemented in `mistral` allow for giving in inputs already evaluated samples (from previous calculation for instance). The formalism is always the following:

- the input samples are given with the variable `X`
- the value of the `lsf` on these samples is given in `y`

`X` should be a matrix with `nrow = dimension` and `ncol = number of samples` so that `length(y) = ncol(X)`. Note that if `y` is missing, it will be calculated by the algorithm.

Short tutorial for using parallel computation with `foreach`

In the previous section we have shown how to use the `foreach` loop to define a well-suited function for using parallel computation. Indeed, `foreach` requires the initilisation of a parallel backend to run *effectively* in parallel. For instance, using the above code without further initilisation will issue a `Warning`:

```
myCode <- function(x) x[1] + x[2]
x <- 1:2
lsf(x)
```

```
## [1] 3
```

Basically, a parallel backend can be understood as a way of defining what *parallel* means for the (master) R session. The simplest, and not of great interest, backend is the *sequential* one:

```
foreach::registerDoSEQ()
```

This tells R that *parallel* means indeed usual sequential computation. However the interest of parallel computation is to run *simultaneously* several tasks.

With R, the management of these *parallel* tasks is let to the task manager of the used computer. In other words, **initialising a parallel backend with R is only a easy way to launch several R sessions and to make them communicate**. This means that there is no theoretical requirement for initialising a backend with a number of parallel *workers* equal to the number of physical cores of the machine. Eventually if more parallel tasks than real cores are initialised, the management of the tasks is let to the native task manager while if less workers are initialised, the `foreach` loop distributes the computational load.

There are two main possible frameworks for parallel computing: OpenMP and MPI. Without digging to much into the details, OpenMP lets you use the several cores of one given *computer* (one shared memory) while MPI allows for using several computers connected with a bus.

Let us first focus on OpenMP. Depending on the OS of the workstation (Windows or Mac/Linux), you can use either `doSNOW` or `doMC`. SNOW (Simple Network of Workstations) is available for both Windows and Unix OS. It requires to first create a cluster with base package `parallel::makeCluster()`. This means that the subsequent R sessions (slave sessions in parallel terminology) are created once for all in the beginning. It is like opening several R sessions by hand: looking at your task manager you will see as many R processes as the size of the requested cluster.

```
# return the number of cores of the computer
n <- parallel::detectCores()
# default behaviour if n not specified explained in the help page
cl <- parallel::makeCluster(1)
doSNOW::registerDoSNOW(cl)

# Control that everything is set properly
foreach::getDoParName()
```

```
## [1] "doSNOW"
```

```
foreach::getDoParWorkers()
```

```
## [1] 1
```

In the end, the cluster has to be closed with:

```
parallel::stopCluster(cl)
```

The other option for Unix OS is `doMC`. The main difference is that the cluster is made by *forking* the current master session, ie. that the sub-sessions are a copy of the current session, including all the variables defined in the `.GlobalEnv`. It is easier to initialise and more robust (SNOW can miss variable even though `foreach` tries an automatic export of the necessary ones):

```
# default behaviour if n not specified explained in the help page
doMC::registerDoMC()
```

Here there is no need to close the cluster because it is created on-the-fly for each instance of the `foreach` loop.

The initialisation of an MPI backend is rather similar to the one of a SNOW backend:

```
# instead of parallel::detectCores() to see the available number of MPI threads
Rmpi::mpi.universe.size()
cl <- doMPI::startMPIcluster()
doMPI::registerDoMPI(cl)
```

and similarly in the end:

```
doMPI::closeCluster(cl)
Rmpi::mpi.quit()
```

The interested reader is referred to the vignettes of the above mentionned packages for further explanations.

Statistical methods for uncertainty quantification

In this section, we describe the *purely* statistical methods proposed in `mistral` for uncertainty quantification. Indeed the uncertainty quantification problem is twofold:

- is there an analytical formula for the sought probability?
- is it possible to use the *real* model `myCode` or is it necessary to build a *surrogate* model?

The statistical methods aim at solving the first issue, ie. at estiamting the probability when no analytical expression is found.

Crude Monte Carlo method

The crude Monte Caro method is based on the Strong Law of Large Numbers. Basically it makes an average of independent and identically distributed (iid) samples. A basic way to implement it could be:

```
X <- matrix(rnorm(2e5), nrow = 2) # generate 1e5 standard Gaussian samples
Y <- mistral::kiureghian(X) # evaluate to model to get 1e5 iid samples
q <- 0 # define the threshold
(p <- mean(Y<q)) # estimate P[g(X)<0]
```

```
## [1] 0.00295
```

The function `mistral::MonteCarlo` is a wrap-up of this simple algorithm. However, instead of specifying a given number of samples, it works iteratively by adding `N_batch` samples per iteration until a target precision `precision` on the probability estimation is reached (usually a coefficient of variation of 5 or 10%) or `N_max` samples have been simulated.

```
mc <- mistral::MonteCarlo(dimension = 2, lsf = mistral::kiureghian, N_max = 1e5, q = q,
                          # these first parameters are exactly the one used above
                           N_batch = 1e4) # define the batch size
```

```
## ========================================================================
##                   Beginning of Monte-Carlo algorithm
## ========================================================================
##
## ========================================================================
##                      End of Monte-Carlo algorithm
## ========================================================================
##
##    - p = 0.00317
##    - q = 0
##    - 95% confidence intervalle : 0.002814475 < p < 0.003525525
##    - cov = 0.0560765
##    - Ncall = 1e+05
```

In this latter example, the target precision is not reached but the algorithm stopped because of the limit given by `N_max`. It is possible to set `N_max = Inf`:

```
mc <- mistral::MonteCarlo(dimension = 2, lsf = mistral::kiureghian, N_max = Inf, q = q,
                          N_batch = 1e4, # define the batch size
                          verbose = 1) # control the level of log messages
```

```
## ========================================================================
##                   Beginning of Monte-Carlo algorithm
## ========================================================================
##
##  * STEP 1 : FIRST SAMPLING AND ESTIMATION
##  * STEP 2 : LOOP UNTIL COV < PRECISION
##  * cov = 0.1793267 > 0.05 and Inf remaining calls to the LSF
##  * cov = 0.1228882 > 0.05 and Inf remaining calls to the LSF
##  * cov = 0.09983319 > 0.05 and Inf remaining calls to the LSF
##  * cov = 0.08824681 > 0.05 and Inf remaining calls to the LSF
##  * cov = 0.07634103 > 0.05 and Inf remaining calls to the LSF
##  * cov = 0.07112999 > 0.05 and Inf remaining calls to the LSF
##  * cov = 0.06655944 > 0.05 and Inf remaining calls to the LSF
##  * cov = 0.06327372 > 0.05 and Inf remaining calls to the LSF
##  * cov = 0.05945577 > 0.05 and Inf remaining calls to the LSF
##  * cov = 0.05581219 > 0.05 and Inf remaining calls to the LSF
##  * cov = 0.05329082 > 0.05 and Inf remaining calls to the LSF
##  * cov = 0.05108295 > 0.05 and Inf remaining calls to the LSF
## ========================================================================
##                      End of Monte-Carlo algorithm
## ========================================================================
##
##    - p = 0.003215385
##    - q = 0
##    - 95% confidence intervalle : 0.002901352 < p < 0.003529418
##    - cov = 0.0488329
```

```
##    - Ncall = 130000
```

and total number of calls (simulated samples) is 1.3×10^5.

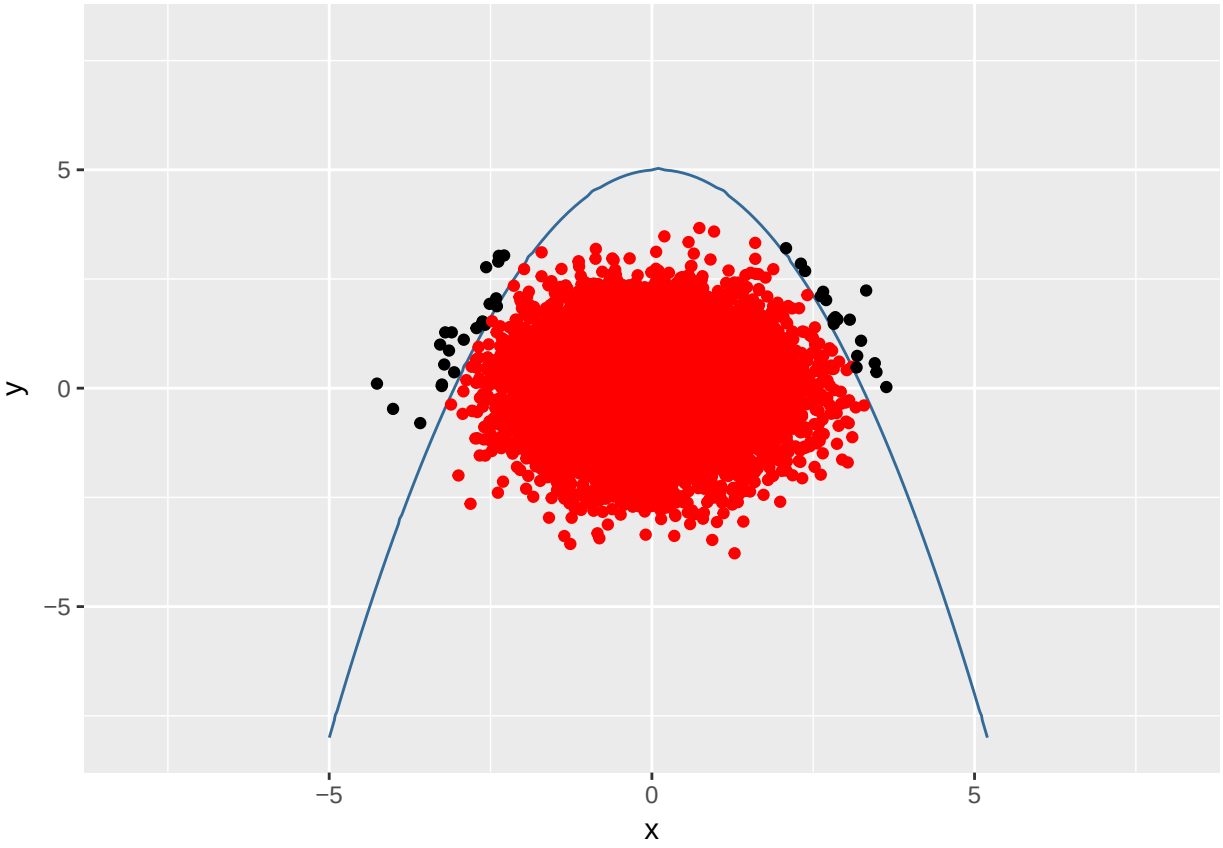
As for the base *cdf* functions `p****` (`pnorm, plnorm, punif...`) the statistical methods of `mistral` integrate a `lower.tail` parameter specifying which tail is to be estimated:

- `lower.tail = TRUE` means that one estimates $P[g(X) < q]$
- `lower.tail = FALSE` means instead $P[g(X) > q]$

For illustrative purpose it is also possible to plot the contour of the limit-state function. All `mistral` functions let draw samples and contour with `ggplot2` functions even though this can be quite memory and time demanding:

```
mc <- mistral::MonteCarlo(dimension = 2, lsf = mistral::kiureghian, N_max = 1e4, q = q,
                          N_batch = 1e4, # define the batch size
                          plot = TRUE)
```

```
## ========================================================================
##                   Beginning of Monte-Carlo algorithm
## ========================================================================
```

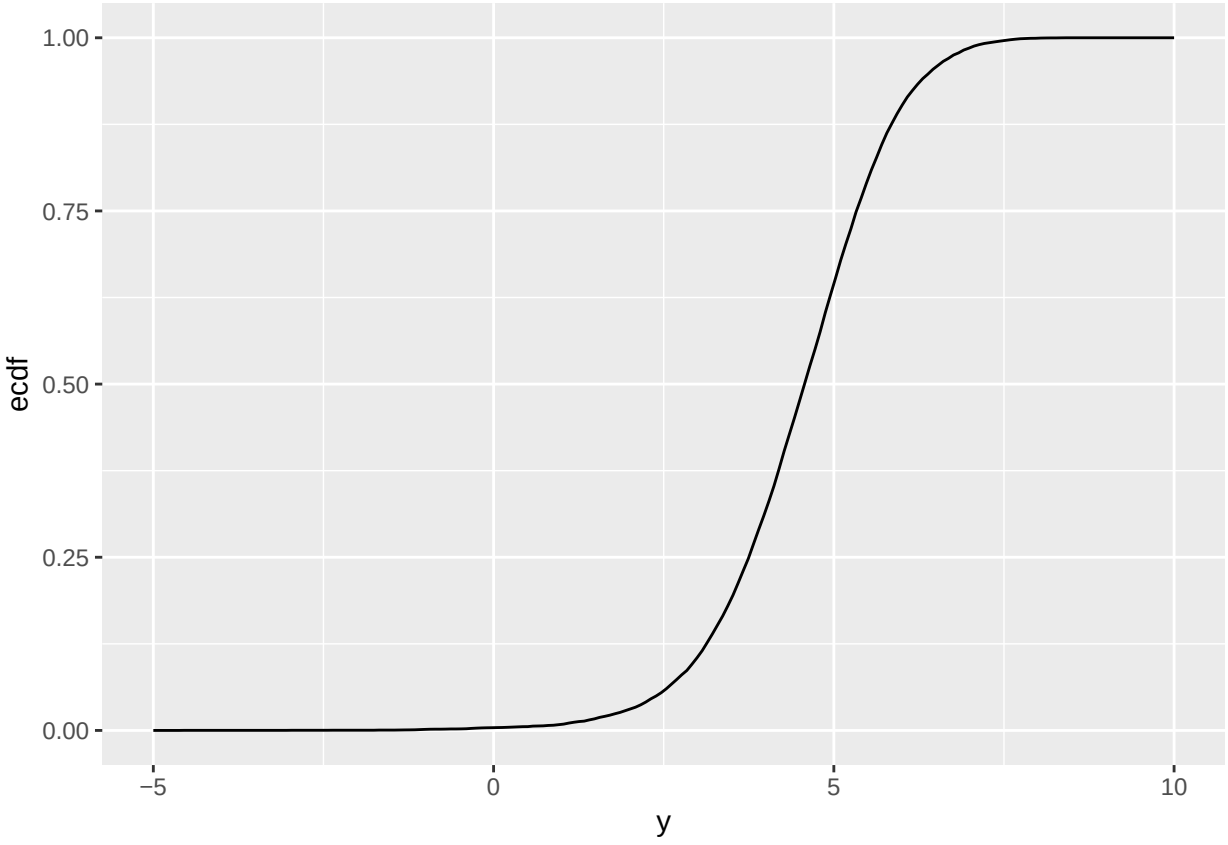



```
## ========================================================================
##                     End of Monte-Carlo algorithm
## ========================================================================
##
##    - p = 0.004
##    - q = 0
##    - 95% confidence intervalle : 0.002737621 < p < 0.005262379
##    - cov = 0.1577973
```

```
##     - Ncall = 10000
```

The `MonteCarlo` method also returns the empirical cdf of the real-valued random variable $Y = g(\mathbf{X})$ (similarly to the base `stats::ecdf` function):

```
require(ggplot2)
y = seq(-5, 10, l = 200)
ggplot(data.frame(y=y, ecdf = mc$ecdf(y)), aes(y,ecdf)) + geom_line()
```



Subset Simulation method

As visible in the Monte Carlo method plot, this method is quite inefficient because it samples a lot in the safety domain (the red dots). To circumvent this limitation, the splitting method, also called *Subset Simulation* and implemented as `mistral::SubsetSimulation` works iteratively on the threshold q. Instead of trying to estimate directly $P[g(X) < q]$ it creates a sequence of intermediate thresholds (q_i) such that the **conditional probabilities are not too small**. Hence, instead of simulating new iid `N_batch` it resamples the `N_batch` using Markov Chain drawing conditionally to be greater than a threshold defined as the `p_0` empirical quantile of the previous batch.

```
ss <- mistral::SubsetSimulation(dimension = 2, lsf = mistral::kiureghian, q = 0,
                                N = 1e4,
                                plot = TRUE)
```

```
## ========================================================================
##                  Beginning of Subset Simulation algorithm
## ========================================================================
##     - q_0 = 2.933605
##     - P = 0.1
```

```
##    - q_0 = 1.045046
##    - P = 0.01

##    - q_0 = 0
##    - P = 0.003063

## ============================================================================
##                  End of Subset Simulation algorithm
## ============================================================================
##    - p = 0.003063
##    - q = 0
##    - 95% confidence intervalle : 0.002787229 < p < 0.003338771
##    - cov = 0.04501641
##    - Ncall = 410000
```

Note here that the total number of calls `Ncall` is much bigger than `3 x N`. Indeed the conditional sampling with Markov Chain drawing requires to retain only one over `thinning = 20` samples. In the end in this example it makes a total of $10^4 + 2 \times 20 \times 10^4 = 410000$. As a matter of fact a naive Monte Carlo estimator with the same computational budget (ie. 410000 samples) would have produced an estimator with a coefficient of variation:

$$cov \approx \sqrt{\frac{1}{p \times 410000}} = 0.0282185$$

As a rule of thumbs when the sought probability is greater than 10^{-3} it is more efficient to use a crude Monte Carlo method than a advanced *Splitting* method[1].

Moving Particles

In the usual Subset Simulation method the cutoff probability for defining the intermediate thresholds is set to `p_0 = 0.1`. Hence at a given iteration `N` samples are generated conditionally greater than this empirical quantile. However, it has been shown that it is statistically optimal (total number of generated samples against coefficient of variation of the final estimator) to resample these `N` *particles* according to **their own level**. It means that instead of using the $N-$sample $(Y_i)_{i=1}^N = (g(\mathbf{X}_i))_{i=1}^N$ to estimate a `p_0` quantile for Y, each $\mathbf{X}_i$ is resampled conditionally to be greater than Y_i.

More precisely, the Moving Particle method aims at simulating independent and identically distributed (iid) realisations of a given random walk over the real-valued output $Y = g(\mathbf{X})$. This random walk is defined as follows: $Y_0 = -\infty$ and

$$Y_{n+1} \sim Y | Y > Y_n$$

In other words, each element is generated conditionally greater than the previous one. This random walk is a Poisson process and lets build true counterparts of the crude Monte Carlo estimators of a probability, of a quantile or of the *cdf* of Y.

Since the algorithm generates iid realisations of such a random walk, it is especially suited for using with parallel computing. Hence the code includes a `foreach` loop and will directely benefit from a parallel backend.

```
foreach::registerDoSEQ()
mp <- mistral::MP(dimension = 2,  lsf = mistral::kiureghian, q = 0, N = 1e2)
```

```
## ============================================================================
##                          Beginning of MP algorithm
## ============================================================================
##  ### PARALLEL PART ###
##  * backend: doSEQ
##  * N.batch = 1
```

[1] this rule depends on the `thinning` parameter.

```
##
## ==========================================================================
##                           End of MP algorithm
## ==========================================================================
##
##    - p = 0.003218816
##    - q = 0
##    - 95% confidence intervalle : 0.002012711 < p < 0.005147672
##    - Total number of calls = 11520
```

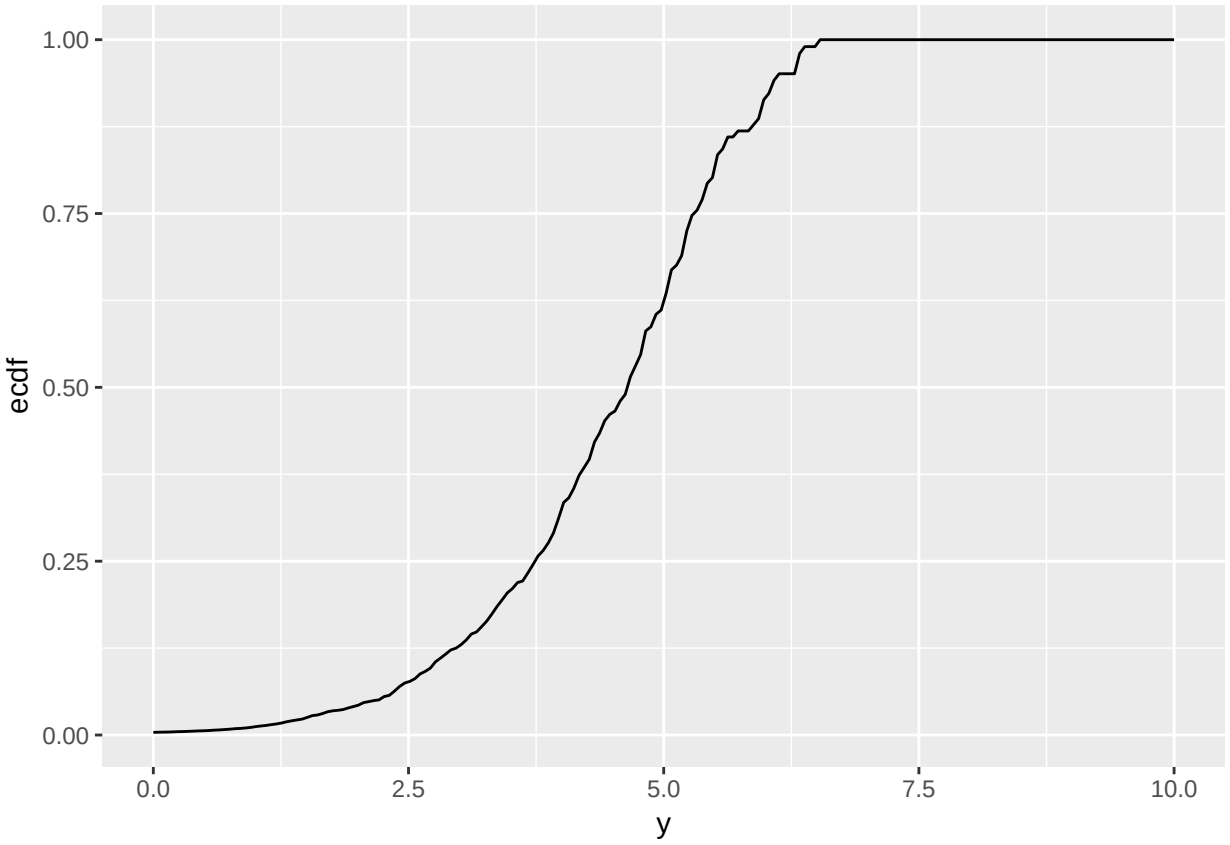
One can compare this result with the one got from `SubsetSimulation` in terms of coefficient of variation against total number of calls.

```
ss$cv^2*ss$Ncall / (mp$cv^2*sum(mp$Ncall))
```

```
## [1] 1.343853
```

The `MP` method not only returns an estimation of the sought probability p but also an empirical *cdf* of $Y = g(\mathbf{X})$ over the interval $[q, \infty)$ (with `lower.tail = TRUE`, the complementary *cdf* over $(-\infty, q]$ otherwise).

```
y = seq(0, 10, l = 200)
ggplot(data.frame(y=y, ecdf = mp$ecdf(y)), aes(y,ecdf)) + geom_line()
```



The empirical *cdf* is vectorized over `q` and returns a `NA` when the value is not in the right interval. It can also be used to estimate a quantile:

```
mp <- mistral::MP(dimension = 2,  lsf = mistral::kiureghian, p = mp$p, N = 1e2)
```

```
## ==========================================================================
##                          Beginning of MP algorithm
```

```
## ============================================================================
##  ============================================
##  STEP 1 : MOVE PARTICLES A GIVEN NUMBER OF TIMES
##  ============================================
##
##  * Number of deterministic event per algorithm = 556
##
##  ### PARALLEL PART ###
##  * backend: doMC
##  * N.batch = 1
##
##  ====================================================
##  STEP 2 : RESTART ALGORITHM UNTIL -0.1564574
##  ====================================================
##
##  ### PARALLEL PART ###
##  * backend: doMC
##  * N.batch = 1
##
## ============================================================================
##                          End of MP algorithm
## ============================================================================
##
##    - p = 0.003857115
##    - q = -0.1473474
##    - 95% confidence intervalle : NA < q < 0.4130965
##    - Maximum number of moves/algorithm = 556
##    - L_max = -0.1564574
##    - Number of moves = 558
##    - Targeted number of moves = 556
##    - Total number of calls = 11240
```

This latter estimation also enables parallel computation but in a 2-step algorithm. The empirical *cdf* is estimated until the farthest *state* reached by the iid random walks. Note also that the condifence interval requires to run the algorithm a little bit longer and thus is optional: default parameter `compute_confidence = FALSE`.

```
mp <- mistral::MP(dimension = 2,  lsf = mistral::kiureghian, p = mp$p, N = 1e2,
                  compute_confidence = TRUE)
```

```
## ============================================================================
##                          Beginning of MP algorithm
## ============================================================================
##  ============================================
##  STEP 1 : MOVE PARTICLES A GIVEN NUMBER OF TIMES
##  ============================================
##
##  * Number of deterministic event per algorithm = 602
##
##  ### PARALLEL PART ###
##  * backend: doMC
##  * N.batch = 1
##
##  ====================================================
##  STEP 2 : RESTART ALGORITHM UNTIL -0.2713296
```

```
##  ====================================================
##
##  ### PARALLEL PART ###
##  * backend: doMC
##  * N.batch = 1
##
## ========================================================================
##                          End of MP algorithm
## ========================================================================
##
##    - p = 0.003857115
##    - q = 0.1943606
##    - 95% confidence intervalle : -0.2691335 < q < 0.6347109
##    - Maximum number of moves/algorithm = 602
##    - L_max = -0.2713296
##    - Number of moves = 604
##    - Targeted number of moves = 603
##    - Total number of calls = 12160
```

Finally, the conditional sampling for the `MP` method also requires Markov Chain drawing and the same disclaimer as for the `SubsetSimulation` applies: this method is really worth for $p \leq 10^{-3}$.

Metamodel based algorithms

In the previous section, we have shown some statistical tools to estimate probability, quantile and *cdf* with a given function g. However, these statistics still require a lot a calls to the model g. Thus it may be necessary to approximate it, ie. to *learn* it with some of its input-output couples $(\mathbf{X}_i, Y_i)_i$.

In `mistral` there are basically three types of metamodel implemented:

- linear model: the `FORM` method *replaces* the model g with a hyperplane crossing the so-called *Most Probable Failure Point*. This point is, in the standard space, the failing sample the closest to the origin.
- Support-Vector Machine (SVM): this classifier is used in the `S2MART` method and relies on the `e1071::svm` function.
- Gaussian process regession; this is used in `MetaIS`, `AKMCS` and `BMP`. Here the model g is *replaced* by a Gaussian random process with known distribution. The regression is carried out using the `DiceKriging` package.

Short tutorial on the metamodels used

Support-Vector Machine

A Support-Vector Machine is a surrogate model which aims at **classifying** the samples of the input space according to some labels attached on it. In our context the label is straightforward: failing or safety domain. From a bunch of input-outputs couples, it builds a frontier and lets classify any new sample $\mathbf{X}$.

When talking about SVM, one often makes mention of the *kernel trick*. Indeed the SVM was initially build as a linear classifier, ie that it looked for an hyperplane separating the input space into two subspaces, each one being related to a given label. However this was far too constraining. The use of a kernel instead of the natural inner product lets build more complex non-linear classifiers.

```
require(e1071)
X <- data.frame(x1 = rnorm(100), x2 = rnorm(100))
Y <- rowSums(X^2)
(svm.model <- svm(X, (Y>1), type = "C-classification"))
```

```
##
## Call:
## svm.default(x = X, y = (Y > 1), type = "C-classification")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1
##       gamma:  0.5
##
## Number of Support Vectors:  45
```

```
X.test <- data.frame(x1=rnorm(1), x2=rnorm(1))
predict(svm.model, X.test)
```

```
##    1
## TRUE
## Levels: FALSE TRUE
```

```
sum(X.test^2)
```

```
## [1] 3.118312
```

The interested reader is referred to the `e1071::svm` vignette for more details about SVM.

Kriging

Kriging refers to the case where the computer code g is seen as a specific realisation of a random process with known distribution.

In computer experiement, this random process is always supposed to be Gaussian. Furthermore, the parameters of the random process covariance are usually estimated with a plug-in approach: in a first step the model is fitted with some data (with Maximum Likelihood Estimation of Cross-Validation for instance). Then they are supposed to be known and thus the distribution of the process at any point $\mathbf{x}$ is Gaussian with known mean and variance.

These quantities are referred to as the kriging mean and the kriging variance. While the first one usually serves as a cheap surrogate model for g, the second one lets characterise the *precision* of the prediction. Especially Kriging interpolates the data: the kriging variance at known location is 0.

```
require(DiceKriging)
X <- data.frame(x1 = rnorm(100), x2 = rnorm(100))
Y <- rowSums(X^2)
km.model <- km(design = X, response = Y)
```

```
##
## optimisation start
## ------------------
## * estimation method   : MLE
## * optimisation method : BFGS
## * analytical gradient : used
## * trend model : ~1
## * covariance model :
##   - type :  matern5_2
##   - nugget : NO
##   - parameters lower bounds :  1e-10 1e-10
##   - parameters upper bounds :  9.77854 14.44376
```

```
##   - best initial criterion value(s) :  434.9003
##
## N = 2, M = 5 machine precision = 2.22045e-16
## At X0, 0 variables are exactly at the bounds
## At iterate     0  f=       -434.9  |proj g|=       1.3168
## At iterate     1  f =      -446.08  |proj g|=       0.75448
## At iterate     2  f =      -446.11  |proj g|=       0.59854
## Nonpositive definiteness in Cholesky factorization in formk;
##    refresh the lbfgs memory and restart the iteration.
## At iterate     3  f =      -446.36  |proj g|=       0.20117
## At iterate     4  f =      -446.38  |proj g|=      0.028276
## At iterate     5  f =      -446.38  |proj g|=      0.001131
## At iterate     6  f =      -446.38  |proj g|=    8.1212e-06
## Bad direction in the line search;
##    refresh the lbfgs memory and restart the iteration.
## Line search cannot locate an adequate point after 20 function
## and gradient evaluations
## final  value -446.382901
## stopped after 6 iterations
```

```
x.new <- data.frame(x1=rnorm(1), x2=rnorm(1))
print(sum(x.new^2))
```

```
## [1] 0.6347545
```

```
predict(km.model, x.new, type = "UK")[c('mean', 'sd')]
```

```
## $mean
## [1] 0.6347515
##
## $sd
## [1] 5.872395e-05
```

The FORM method

The `mistral::FORM` function always tries to estimate $P[g(\mathbf{X}) < 0]$ with $\mathbf{X}$ in the standard space. As for statistical methods, `mistral::UtoX` can be used to transform the original limit-state function onto a suitable one.

Furthermore the limit-state function may have to be modified to fit the used framework: say for instance that one wants to estimate $P[g(\mathbf{X}) > q]$, then one should define:

```
lsf.FORM = function(x) {
   q - g(x)
}
```

The `FORM` function requires two parameters: a starting point for the research of the Most Probable Failing Point `u.dep` and a total number of calls `N.calls`:

```
form <- mistral::FORM(dimension = 2, mistral::kiureghian, N.calls = 1000, u.dep = c(0,0))
form$p
```

```
## [1] 0.001832186
```

The FORM method gives an analytical expression of the sought probability replacing the true model g with the found hyperplane. However, `mistral::FORM` also implements an Importance Sampling scheme. Instead

of using this ready-made formula, it makes an IS estimation with a Gaussian standard proposal distribution centred at the MPFP.

```
form.IS <- mistral::FORM(dimension = 2, mistral::kiureghian, N.calls = 1000,
                         u.dep = c(0,0),
                         IS = TRUE)
form.IS$p
```

```
## [1] 0.0018599
```

In this latter case, the variance and the confidence interval at 95% are given in output. Note however that the estimated variance may be far from the real one.

The `MetaIS` method

`MetaIS`, for Metamodel-based Importance Sampling, is another metamodelling technique using a surrogate model in addition to an importance sampling scheme. Here, instead of using the input distribution only re-centred at the MPFP, the optimal (zero-variance) importance distribution is approximated with a Kriging-based surrogate model.

More precisely, recall that the optimal distibution is given by:

$$\pi(\mathbf{x}) = \frac{1_{g(\mathbf{x})>q}}{P[g(\mathbf{X}) > q]}$$

the *hard* indicator function $1_{g(\mathbf{x})>q}$ is *replaced* by its kriging counterpart:

$$\tilde{\pi}(\mathbf{x}) = P[\xi(\mathbf{x}) > q]$$

where ξ is the Gaussian process modelling the uncertainty on the computer code g. With the Gaussian hypothesis, its distribution is known.

The algorithm is then twofold: first the Gaussian process is *learnt*, with means that input-output samples are calculated to get a conditional distribution of the process. Then a usual Importance Sampling scheme is run. The points added iteratively to the Design of Experiments (DoE) are chosen by clustering samples generated into the *margin*. When several calls to g can be made in parallel, several points can then be added to the DoE simultaneously by chosing the number of cluster `K_alphaLOO` accordingly.

The enrichment step stops either when the stopping criterion is reached or when the total given nummber of samples is simulated. Then few other calls to g have to made for the Importance Sampling estimator.

```
metais <- mistral::MetaIS(dimension = 2, lsf = mistral::waarts, N = 3e5, K_alphaLOO = 5,
                          plot = TRUE)
```

```
## ========================================================================
##                       Beginning of Meta-IS algorithm
## ========================================================================
##
## ========================================================================
##  STEP 1 : Adaptative construction of h the approximated optimal density
## ========================================================================

##
##  A- REFINEMENT OF PROBABILISTIC CLASSIFICATION FUNCTION PI
##     ======================================================
##
## FIRST DoE
##  -------------
```

```
## ITERATION  1
##  -------------

## ITERATION  2
##  -------------

## ITERATION  3
##  -------------

## ITERATION  4
##  -------------

## ITERATION  5
##  -------------

## ITERATION  6
##  -------------

##
##  B- ESTIMATE AUGMENTED FAILURE PROBABILITY USING MC ESTIMATOR
##     ========================================================
##  P_epsilon = 0.002342403
##  cov_epsilon = 0.03573595
##
## ================================================
##  STEP 2 : Adaptative importance sampling scheme
## ================================================

##  alpha = 1.085359
##  cov_alpha = 0.03110854
## ============================================================================
##                      End of Meta-IS algorithm
## ============================================================================
##
##    - P_epsilon = 0.002342403
##    - 95% conf. interv. on P_epsilon: 0.002174987 < p < 0.002564427
##    - alpha = 1.085359
##    - 95% conf. interv. on alpha: 1.017831 < alpha < 1.152887
##    - p = 0.002542348
##    - 95% conf. interv. on p: 0.002301372 < p < 0.002783324
```

The AKMCS method

AKMCS, for Active learning using Kriging an Monte Carlo Simulation, is an other kriging-based approach. Instead of using the Gaussian process to define a Importance density, it uses the Kriging mean as a cheap surrogate for the computer code g in a crude Monte Carlo estimator.

The originality and the efficiency of the AKMCS method comes from the fact that it samples **from the beginning** the Monte Carlo population and then focus on *learning* this population instead of the whole input space. The learning step is then an iterative search of the *more uncertain* points.

Note however that this discretisation makes the algorithm generating quite huge matrices and can lead to memory issues for extreme probabilities $p \leq 10^{<-5}$.

```
akmcs <- mistral::AKMCS(dimension = 2, lsf = mistral::waarts, N = 3e5, plot = TRUE, Nmax = 10)
```

```
## ============================================================================
##                     Beginning of AK-MCS algorithm
```

```
## ========================================================================

##  ==========================================
##  STEP 1 : GENERATION OF THE WORKING POPULATION
##  ==========================================
##
##  ==================
##  STEP 2 : FIRST DoE
##  ==================

## Warning: Quick-TRANSfer stage steps exceeded maximum (= 15000000)

##   - minimum value of the criterion = 2.031557 estimated in 4.856 sec. with 1 worker(s)

## Warning: Not possible to generate contour data

## Warning: Not possible to generate contour data

##  ======================
##  STEP 3 : UPDATE THE DoE
##  ======================

## Warning: Not possible to generate contour data

## Warning: Not possible to generate contour data

##    - minimum value of the criterion = 0.003241411 estimated in 4.836 sec. with 1 worker(s)

##    - minimum value of the criterion = 0.002755663 estimated in 4.36 sec. with 1 worker(s)

##    - minimum value of the criterion = 0.0006564013 estimated in 4.898 sec. with 1 worker(s)

##    - minimum value of the criterion = 0.0001908488 estimated in 4.313 sec. with 1 worker(s)

##    - minimum value of the criterion = 0.002998511 estimated in 5.366 sec. with 1 worker(s)

##    - minimum value of the criterion = 0.009833194 estimated in 4.645 sec. with 1 worker(s)

##    - minimum value of the criterion = 0.01585155 estimated in 4.885 sec. with 1 worker(s)

##    - minimum value of the criterion = 0.006862492 estimated in 4.65 sec. with 1 worker(s)

##    - minimum value of the criterion = 0.05930103 estimated in 4.727 sec. with 1 worker(s)

##    - minimum value of the criterion = 0.06011172 estimated in 5.522 sec. with 1 worker(s)

##  ====================================
##  STEP 4 : EVALUATE FAILURE PROBABILITY
##  ====================================
##    - p = 0.001233612
##    - failure = 0
##    - 95% confidence interval on Monte Carlo estimate: 0.001105441 < p < 0.001361783
##  * cov = 0.05194958
##    => cov too large ; this order of magnitude for the probability brings N = 323852
```

The BMP method

The BMP method is a Bayesian version of the previous MP algorithm. Indeed, in this algorithm, the code g is considered as a Gaussian random process ξ. Given a database of input-ouput couples, it is possible to estimate probability, quantile and moment of the *augmented* real-valued random variable $Y = \xi(\mathbf{X})$:

- $\mathbf{X}$ is the random vector of intputs
- ξ is a Gaussian random process embedding the uncertainty on the code g

- $Y|\mathbf{X}$ is a Gaussian random variable
- $P[Y > q] = \int_{\mathbb{X}} P[Y > q|\mathbf{X} = \mathbf{x}] d\mu^X(\mathbf{x})$

For instance one can use the database created by the `AKMCS` method stored in the output `akmcs`:

```
bmp <- mistral::BMP(dimension = 2, lsf = mistral::waarts, q = 0, N = 100,
                    N.iter = 0, X = akmcs$X, y = akmcs$y)
```

```
## ========================================================================
##                 Beginning of BMP algorithm
## ========================================================================
##  * parallel backend registered: FALSE
##  * parallel backend version:
##    - number of workers: 1
##  ============================================================
##  BEGINNING : FIRST DoE given in inputs: 30 samples
##  ============================================================
##
##  * 30 points added to the model in 0.209 sec
##    - covtype = matern5_2
##    - coef.cov = 1.53206 1.789638
##    - coef.var = 0.7910187
##    - coef.trend = 0
##  ================================================
##  ENRICHMENT STEP: 0 samples to be added to the DoE
##  ================================================
##
##  * Final Poisson process N = 100 generated in 8.266 sec with 1 workers
## ========================================================================
##                       End of BMP algorithm
## ========================================================================
##
##  * Current alpha estimate = 0.001276833
##  * Current cv estimate = 0.2581351
##  * Current h estimate = 0.1682699
##  * Current I estimate = 1.411728
##    - alpha = 0.001276833
##    - cv = 0.2581351
##    - q = 0
##    - 95% confidence intervalle : 0.0007619399 < alpha < 0.002139673
##    - Total number of calls = 30
```

The method can also be used, as the other one, to *learn* the metamodel:

```
bmp <- mistral::BMP(dimension = 2, lsf = mistral::waarts, q = -4, N = 100,
                    N.iter = 2, plot = TRUE)
```

```
## ========================================================================
##                 Beginning of BMP algorithm
## ========================================================================
##  * parallel backend registered: FALSE
##  * parallel backend version:
##    - number of workers: 1
##  ============================================================
##  BEGINNING : FIRST DoE with uniform design: 10 samples
##  ============================================================
```

```
##
##  * 10 points added to the model in 0.054 sec
##    - covtype = matern5_2
##    - coef.cov = 4.924931 4.742282
##    - coef.var = 5.120157
##    - coef.trend = 4
##  ================================================
##  ENRICHMENT STEP: 2 samples to be added to the DoE
##  ================================================
##
##  Remaining iter. : 2
##  * Poisson process N = 100 generated in 8.629 sec with 1 workers
##  * Current alpha estimate = 4.365349e-06
##  * Current cv estimate = 0.3513092
##  * Current h estimate = 0.9487341
##  * Current I estimate = 8.104428
##  * Evaluation of SUR criterion: integrated = TRUE, r = 1, approx = FALSE, approx.pnorm = FALSE, optir
##  * SUR criterion: 1328 points tested in 32.089 sec
##  * Call the lsf on the proposed point(s)
##  * Lsf evaluated in 0 sec
##  * 1 points added to the model in 0.063 sec
##    - covtype = matern5_2
##    - coef.cov = 5.244798 5.874224
##    - coef.var = 6.246869
##    - coef.trend = 4
##  Remaining iter. : 1
##  * Poisson process N = 100 generated in 8.376 sec with 1 workers
##  * Current alpha estimate = 2.836559e-07
##  * Current cv estimate = 0.3882719
##  * Current h estimate = 0.9511388
##  * Current I estimate = 8.927449
##  * Evaluation of SUR criterion: integrated = TRUE, r = 1, approx = FALSE, approx.pnorm = FALSE, optir
##  * SUR criterion: 1600 points tested in 45.763 sec
##  * Call the lsf on the proposed point(s)
##  * Lsf evaluated in 0 sec
##  * 1 points added to the model in 0.048 sec
##    - covtype = matern5_2
##    - coef.cov = 4.295536 3.166985
##    - coef.var = 4.092989
##    - coef.trend = 4
## Warning: Not possible to generate contour data
##  * Final Poisson process N = 100 generated in 9.524 sec with 1 workers
## ================================================================================
##                        End of BMP algorithm
## ================================================================================
##
##  * Current alpha estimate = 2.228623e-07
##  * Current cv estimate = 0.3913657
##  * Current h estimate = 0.9591293
##  * Current I estimate = 10.13233
##    - alpha = 2.228623e-07
##    - cv = 0.3913657
```

```
##    - q = -4
##    - 95% confidence intervalle : 1.018827e-07 < alpha < 4.874976e-07
##    - Total number of calls = 12
```

In this latter case, the learning step is driven with the minimisation at each iteration of the integrated criterion I. This can be chosen with the argument `sur`. This integrated criterion is especially interesting when the sought probability is extreme. In this context, `BMP` is a preferred alternative to `S2MART`.

Note also that the estimation of quantities h and I can also be done in `AKMCS` and `MetaIS` at each iteration with their parameter `sur=TRUE` (default is `sur=FALSE`) to monitor the learning of the Gaussian process and to compare the different learning strategies.

Conclusion

In this vignette we wanted to present some of the methods implemented in `mistral` for reliability analysis. Precisely we focused on methods designed to estimate a probability of the code exceeding a given threshold.

In this setting, all the statistical and/or metamodel-based algorithms have proven in some cases good efficiency, though some of them are quite time consumming. In this context it should be remembered that they are defined considering that the calls to the limit-state functions are the only important parts while it can appear indeed that all the *side computations* take indeed much more time. The method should then be choosen accordingly. Especially in this vignette some parameters are very low to save computational time on laptop and should be increased for real experiments (see `S2MART` for instance).

Amongst all the strategies proposed in this package, the `Moving Particles` and `Bayesian Moving Particles` are the only one not only focusing on the given threshold but delivering an uncertainty quantification for the random output *until* this threshold. This means that the output is not a given probability but an estimation of the *cdf* of the unknown real-valued random variable.

The interested reader is referred to the academic papers for a deeper understanding of the algorithms and to the package documentation for a comprehensive list of the involved parameters. “‘