Available online at www.eccomasproceedia.org
Eccomas Proceedia UNCECOMP (2023) 419-436

ECCOMAS
Proceedia

UNCECOMP 2023
5th ECCOMAS Thematic Conference on
Uncertainty Quantification in Computational Science and Engineering
M. Papadrakakis, V. Papadopoulos, G. Stefanou (eds.)
Athens, Greece, 12-14 June 2023

UNCERTAINTYQUANTIFICATION.JL: A NEW FRAMEWORK FOR UNCERTAINTY QUANTIFICATION IN JULIA

Jasper Behrensdorf[1], Ander Gray[2], Matteo Broggi[1], and Michael Beer[1,3,4]

[1]Institute for Risk and Reliability, Leibniz University Hannover, Hannover, Germany
behrensdorf,broggi,beer@irz.uni-hannover.de

[2]United Kingdom Atomic Energy Authority, Culham, UK
ander.gray@ukaea.uk

[3]Institute for Risk and Uncertainty, University of Liverpool, Liverpool, UK

[4]International Joint Research Center for Engineering Reliability and Stochastic Mechanics, Tongji University, Shanghai, China

Abstract. *This work presents a new framework for uncertainty quantification developed as a package in the Julia programming language called UncertaintyQuantification.jl. Julia is a modern high-level dynamic programming language ideally suited for tasks like data analysis and scientific computing. UncertaintyQuantification.jl was developed from the ground up to be generalized and flexible while at the same time being easy to use. Leveraging the features of a modern language such as Julia allows to write efficient, fast and easy to read code. Especially noteworthy is Julia's core feature multiple dispatch which enables us to, for example, develop methods with a large number of varying simulation schemes such as standard Monte Carlo, Sobol sampling, Halton sampling, etc., yet minimal code duplication. Current features of UncertaintyQuantification.jl include simulation based reliability analysis using a large array of sampling schemes, local and global sensititivity analysis, meta modelling techniques such as response surface methodology or polynomial chaos expansion as well as the connection to external solvers by injecting values into plain text files as inputs. Through Julia's existing distributed computing capabilities all available methods can be easily run on existing clusters with just a few lines of extra code.*

Keywords: Uncertainty Quantification, Julia, Software, Simulation

ISSN:2623-3339
Peer-review under responsibility of the organizing committee of UNCECOMP 2023.
doi: 10.7712/120223.10347.19810

1 INTRODUCTION

The increasing complexity of modern engineering systems requires adequate simulation methods to ensure their safety and reliability. At the same time, a reliable analysis can only be performed when taking into consideration the present uncertainties. New analyses and simulation methods for the treatment of these uncertainties are constantly developed. A generic and freely available framework, which includes these methods and can be applied to a large array of engineering problems, can be a valuable tool for researchers, students, and industry alike.

Many such frameworks have been developed for a variety of programming languages. *Dakota* (C++) [1], *OpenTURNS* (C++/Python) [2], *OpenCossan* (MATLAB) [3], *UQLab* (MATLAB) [4], and *UQpy* (Python) [5] to name a few.

This work presents an alternative to these frameworks called *UncertaintyQuantification.jl* [6]. This new framework is developed as a module for the Julia programming language. The Julia language [7] has been developed as a new approach to scientific computing. Julia is designed to be fast yet easy to use at the same time. Its core principal *multiple dispatch* allows to write simple reusable code, where the appropriate algorithm is automatically selected based on the input arguments to a function. This allows us to efficiently implement a variety of simulation techniques for the same underlying analysis with little extra effort. Julia was designed from the ground up with parallelization and high performance computing in mind. The large numerical demand of modern simulation methods can often be distributed to computing clusters with only a few extra lines. This makes Julia ideally suited for the development of a generic and complete package for uncertainty quantification.

UncertaintyQuantification is registered in the official *General* registry and can be installed from the Julia REPL by switching to the package manager with a closing square bracket `]`.

```
julia> ] add UncertaintyQuantification
```

The goal of *UncertaintyQuantification* is to build a complete and generalized framework for uncertainty quantification where newly developed methods are constantly implemented and shared with fellow researchers. *UncertaintyQuantification* is released under the MIT license and publicly available on Github. While development is still in the early stages, the basic features and key analyses have been completed. In its current version *UncertaintyQuantification* includes simulation based reliability analysis through standard and advanced Monte Carlo simulation, local and global sensitivity analysis, metamodels, and the ability to connect any of these methods to external solvers. This paper only presents the basic usage of the module. For a thorough explanation of all included simulations the reader is refered to the online documentation.

The remainder of the paper is structured as follows. In Section 2 we introduce the basics of *UncertaintyQuantification* including how to define input parameters, random variables and models. Section 3 and Section 4 present how to use the framework to perform reliability and sensitivity analyses, respectively. How to build metamodels as approximations is presented in Section 5, followed by a few numerical examples in Section 6. The paper closes with conclusions and a brief outlook into future development of the framework.

2 GETTING STARTED

In this section we introduce the basic building blocks of *UncertaintyQuantification*. This includes the inputs such as `Parameter` or `RandomVariable` which will feed into any `Model` for a variety of different analyses. We will also present more advanced concepts including how to model dependencies between the inputs through copulas.

2.1 Inputs

2.1.1 Parameters

A `Parameter` is defined as a constant scalar value. In addition to value the constructor also requires a `Symbol` by which it can later be identified in the `Model`. A `Symbol` is a Julia object which is often used as a name or label. `Symbol`s are defined using the `:` prefix. Parameters represent constant deterministic values. As an example we define a `Parameter` representing the gravity of Earth.

```
g = Parameter(9.81, :g)
```

Parameters are very handy when constants show up in the `Model` in multiple spaces. Instead of updating every instance in the `Model`, we can conveniently update the value by changing a single line.

2.1.2 Random Variables

A `RandomVariable` is essentially a wrapper around any `UnivariateDistribution` defined in the *Distributions.jl* package [8]. Similarly to the `Parameter`, the second argument to the constructor is a `Symbol` acting as a unique identifier. For example, a standard gaussian random variable is defined by passing `Normal()` and `:x` as arguments.

```
x = RandomVariable(Normal(), :x)
```

A list of all possible distributions can be generated by executing `subtypes(UnivariateDistribution)` in the Julia REPL (read-eval-print loop). Note that, *Distributions* is re-exported from *UncertaintyQuantification* and no separate `using` statement is necessary. In addition, the most important methods of the `UnivariateDistribution` including `pdf`, `cdf`, and `quantile`, are also defined for the `RandomVariable`.

Random samples can be drawn from a `RandomVariable` by calling the `sample` method passing the random variable and the desired number of samples.

```
samples = sample(x, 100) # sample(x, MonteCarlo(100))
```

The `sample` method returns a `DataFrame` with the samples in a single column. When sampling from a `Vector` of random variables these invidivual columns are automatically merged into one unified `DataFrame`. By default, this will use stardard Monte Carlo simulation to obtain the samples. Alternatively, any of the quasi-Monte Carlo methods can be used instead.

```
samples = sample(x, SobolSampling(100))
samples = sample(x, LatinHypercubeSampling(100))
```

Many of the advanced simulations, e.g. line sampling [9] or subset simulation [10] require mappings to (and from) the standard normal space, and these are exposed through the `to_standard_normal_space!` and `to_physical_space!` methods respectively. These operate on a `DataFrame` and as such can be applied to samples directly. The transformation is done in-place, i.e. no new `DataFrame` is returned. As such, in the following example, the samples end up exactly as they were in the beginning.

```
to_standard_normal_space!(x, samples)
to_physical_space!(x, samples)
```

2.1.3 Dependencies

UncertaintyQuantification supports modelling of dependencies through copulas. By using copulas, the modelling of the dependence structure is separated from the modelling of the univariate marginal distributions. The basis for copulas is given by Sklar's theorem [11]. It states that any multivariate distribution H in dimensions $d \geq 2$ can be separated into its marginal distributions F_i and a copula function C.

$$H(x_1, \ldots, x_2) = C(F_1(x_1), \ldots, F_d(x_d)) \tag{1}$$

For a thorough discussion of copulas, see [12].

In line with Sklar's theorem we build the joint distribution of two dependent random variables by separately defining the marginal distributions.

```
x = RandomVariable(Normal(), :x)
y = RandomVariable(Uniform(), :y)
marginals = [x, y]
```

Next, we define the copula to model the dependence. *UncertaintyQuantification* supports Gaussian copulas for multivariate ($d \geq 2$) dependence as well as a list of Archimedean copulas for bivariate dependence. The Archimedean copulas are implemented as a wrapper around the *BivariateCopulas.jl* [13] package. Here, we define a Gaussian copula by passing the correlation matrix and then build the `JointDistribution` from the copula and the marginals.

```
cop = GaussianCopula([1 0.8; 0.8 1])
joint = JointDistribution(marginals, cop)
```

Figure 1 shows the samples drawn by sampling from the independent marginals and the joint distribution using the Gaussian copula.

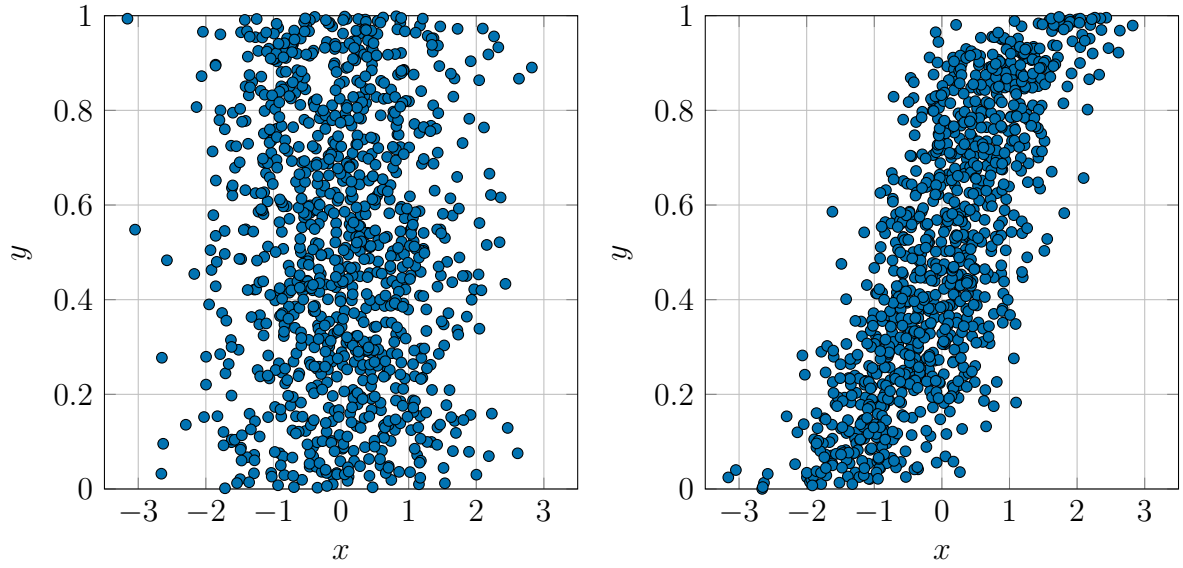
2.2 Models

In this section we present the models included in *UncertaintyQuantification*. A model, in its most basic form, is a relationship between a set of input variables $x \in \mathbb{R}^{n_x}$ and an output $y \in \mathbb{R}$. Currently, most models are assumed to return single-valued outputs. However, as seen later, the `ExternalModel` is capable of extracting an arbitrary number of outputs from a single run of an external solver.

2.2.1 Model

A `Model` is essentially a native Julia function operating on the previously defined inputs. Building a `Model` requires two things: a `Function`, which is internally passed a `DataFrame` containing the samples and must return a `Vector` containing the model response for each sample, and a `Symbol` which is the identifier used to add the model output into the `DataFrame`.

Suppose we wanted to define a `Model` which computes the distance from the origin of two variables x and y as z. We first define the function and then pass it to the `Model`.



(a) Independent

(b) Dependent

Figure 1: 1000 samples drawn from the independent marginal distributions (a) and the dependent joint distribution (b).

```
function z(df::DataFrame)
  return @. sqrt(df.x^2 + df.y^2)
end
m = Model(z, :z)
```

An alternative for a simple model such as this, is to directly pass an anonymous function to the `Model`.

```
m = Model(df -> sqrt.(df.x.^2 .+ df.y.^2), :z)
```

After defining it, a `Model` can be evaluated on a set of samples by calling the `evaluate!` method. This will add the model outcome to the `DataFrame`. Alternatively, the reponse can be obtained as a vector, by calling the `Model` as a function.

```
samples = sample([x, y], MonteCarlo(1000))
evaluate!(m, samples) # add to the DataFrame
output = m(samples) # return a Vector
```

However, most of the time manual evaluation of the `Model` will not be necessary as it is done internally by whichever analysis is performed.

2.2.2 ParallelModel

With the basic `Model` it is up to the user to implement an efficient function which returns the model responses for all samples simultaneously. Commonly, this will involve vectorized operations as presented in the example. For more complex or longer running models, *UncertaintyQuantification* provides a simple `ParallelModel`. This model relies on the capabilites of the *Distributed* module, which is part of the standard library shipped with Julia. Without any present *workers*, the `ParallelModel` will evaluate its function in a loop for each sample. If one

or more workers are present, it will automatically distribute the model evaluations. For this to work, *UncertaintyQuantification* must be loaded with the `@everywhere` macro in order to be loaded on all workers. In the following example, we first load *Distributed* and add four local workers. A simple model is then evaluated in parallel. Finally, the workers are removed.

```
using Distributed
addprocs(4) # add 4 local workers

@everywhere using UncertaintyQuantification

x = RandomVariable(Normal(), :x)
y = RandomVariable(Normal(), :y)

m = ParallelModel(df -> sqrt(df.x^2 .+ df.y^2), :z)

samples = sample([x, y], 1000)
evaluate!(m, samples)

rmprocs(workers()) # release the local workers
```

It is important to note, that the `ParallelModel` requires some overhead to distribute the function calls to the workers. Therefore it performs significantly slower than the standard `Model` with vectorized operations for a simple function as in this example.

By using *ClusterManagers.jl* [14] to add the workers, the `ParallelModel` can easily be run on an existing compute cluster such as *Slurm*.

2.2.3 ExternalModel

The `ExternalModel` provides interaction with almost any third-party solver. The only requirement is, that the solver uses text-based input and output files in which the values sampled from the random variables can be injected for each individual run. The output quantities are then extracated from the files generated by the solver using one (or more) `Extractor`(s). This way, the simulation techniques included in this module, can be applied to advanced models in finite element software such as *OpenSees* or *Abaqus*.

The first step in building the `ExternalModel` is to define the folder where the source files can be found as well as the working directory. Here, we assume that the source file for a simple supported beam model is located in a subdirectory of our current working directory. Similarly, the working directory for the solver is defined. In addition, we define the exact files where values need to be injected, and any extra files required. No values will be injected into the files specified as extra. In this example, no extra files are needed, so the variable is defined as an empty `String` vector.

```
sourcedir = joinpath(pwd(), "demo/models")
sourcefiles = ["supported-beam.tcl"]
extrafiles = String[]
workdir = joinpath(pwd(), "supported-beam")
```

Next, we must define where to inject values from the random variables and parameters into the input files. For this, we make use of the *Mustache.jl* [15] and *Formatting.jl* [16] modules.

The values in the source file must be replaced by triple curly bracket expressions of the form `{{{ :x }}}`, where `:x` is the identifier of the `RandomVariable` or `Parameter` to be injected. For example, to inject the Young's modulus and density of an elastic isotropic material in *OpenSees*, one could write the following.

```
nDMaterial ElasticIsotropic 1 {{{ :E }}} 0.25 {{{ :rho }}}
```

This identifies where to inject the values, but not in which format. For this reason, we define a `Dict{Symbol, String}` which maps the identifiers of the inputs to a Python-style format string. In order to inject our values in scientific notation with eight digits, we use the format string `".8e"`. For any not explicitly defined `Symbol` we can include `:*` as a fallback.

```
formats = Dict(:E => ".8e",:rho => ".8e", :* => ".12e")
```

After formatting and injecting the values into the source file, it would look similar to this.

```
nDMaterial ElasticIsotropic 1 9.99813819e+02 0.25 3.03176259e+00
```

Now that the values are injected into the source files, the next step is to extract the desired output quantities. This is done using an `Extractor`. The `Extractor` is designed similarly to the `Model` in that it takes a `Function` and a `Symbol` as its parameters. However, where a `DataFrame` is passed to the `Model`, the working directoy for the currently evaluated sample is passed to the function of the `Extractor`. The user defined function must then extract the required values from the file and return them. Here, we make use of the *DelimitedFiles* module to extract the maximum absolute displacement from the output file that *OpenSees* generated.

```
disp = Extractor(base -> begin
  file = joinpath(base, "displacement.out")
  data = readdlm(file, ' ')

  return maximum(abs.(data[:, 2]))
end, :disp)
```

An arbitrary number of `Extractor` functions can be defined in order to extract multiple output values from the solver.

The final step before building the model is to define the solver. The solver requires the path to the binary, and the input file. Optional command line arguments can be passed to the `Solver` through the `args` keyword. If the solver binary is not on the system path, the full path to the executable must be defined. Finally, the `ExternalModel` is assembled.

```
opensees = Solver(
  "OpenSees",
  "supported-beam.tcl";
  args = ""
)

ext = ExternalModel(
  sourcedir, sourcefiles, disp, opensees; formats=numberformats,
  ↪  workdir=workdir, extras=extrafiles
)
```

A full example of how to run a reliability analysis of a model defined in *OpenSees* can be found in the demo files of *UncertaintyQuantification*.

3 RELIABILITY ANALYSIS

Reliability analysis, as in the computation of the probability of failure of an engineering system, is facilitated through the simple method interface `probability_of_failure`. The method used to calculate this failure probability is selected internally based on the supplied arguments making use of Julia's multiple dispatch.

All methods to compute the probability of failure require the definition of a performance (limit-state) function $g(x)$ where $x \in \mathbb{R}^{n_x}$ is the set of input variables. By definition, failure of the system occurs for $g(x) < 0$. When implementing the limit-state, the Julia function operates on a `DataFrame` as seen with the `Model`. The `DataFrame` passed to the performance function includes the responses of all models.

3.1 First order reliability method

In the *first-order reliability method* (FORM) [17], [18], the random variables are transformed into the uncorrelated standard normal space and the limit-state function is approximated by a first-order Taylor series expansion at the design point. The design point being the point on the limit-state closest to the origin in the standard normal space.

Assume two random variables $x_1 \sim N(200, 20)$ and $x_2 \sim N(150, 10)$ with the failure domain being

$$F = \{(x_1, x_2) : x_2 > x_1\}. \tag{2}$$

The following script will compute the failure probability using FORM. The solution is obtained using the Hasofer-Lind-Rackwitz-Fiessler (HL-RF) algorithm.

```
form = FORM()

x1 = RandomVariable(Normal(200, 20), :x1)
x2 = RandomVariable(Normal(150, 10), :x2)

function g(df)
  return df.x1 .- df.x2
end

pf, beta, dp = probability_of_failure(g, [x1, x2], form)
```

The returned quantities are the probability of failure $p_f \approx 0.012673$, the reliability index $\beta \approx 2.23606$ and the design point $dp \approx (160, 160)$. In this case, the `probability_of_failure` function is called without a separate model, which can be included as the first input if desired. The `FORM` struct holds no information and its sole purpose is to dispatch to the correct method.

Currently *UncertaintyQuantification* only implements the HL-RF algorithm to compute the design point. To circumvent the algorithm's known shortcomings, we plan to add an alternative method, where the design point identification is solved as an optimization problem, in the future.

3.2 Monte Carlo Simulation

The probability of failure can also be approximated by a variety of Monte Carlo simulation techniques. The module includes standard Monte Carlo simulation as well as a number of quasi-Monte Carlo simulation methods such as Latin hypercube sampling or Sobol' sampling. In order to compute the probability of failure for the simple example presented in the previous subsection through standard Monte Carlo simulation, we need only replace the `FORM` struct with a `MonteCarlo` struct. The only parameter necessary to construct the `MonteCarlo` object is the desired number of samples to be used.

```
mc = MonteCarlo(10000)
pf, c, samples = probability_of_failure(g, [x1, x2], mc)
```

The probability of failure obtained from the simulation is $p_f = 0.0129$ with a coefficient of variation of 0.87475 which agrees with the results obtained by FORM in the previous section. In difference to FORM, any of the Monte Carlo methods will return the the coefficient of variation and the evaluated samples instead of the reliability index and the design point.

To run the simulation with quasi-Monte Carlo techniques, simply replace the `MonteCarlo` struct.

```
pf, c, samples = probability_of_failure(g, [x1, x2],
↪  LatinHypercubeSampling(10000))
pf, c, samples = probability_of_failure(g, [x1, x2], HaltonSampling(10000))
pf, c, samples = probability_of_failure(g, [x1, x2], SobolSampling(10000))
```

UncertaintyQuantification provides some advanced Monte Carlo methods, namely `LineSampling` and `SubSetSimulation`. For a more complex example using `SubSetSimulation` to estimate small failure probabilies, see Section 6.1.

4 SENSITIVITY ANALYSIS

This section presents the methods for both local and global sensitivity analysis included in *UncertaintyQuantification*. Sensitivity analysis examines how the uncertainty of the model output can be attributed to the uncertainty of the inputs [19]. There are two major areas of sensitivity. Where local sensitivity analysis considers the influence of inputs in a certain point, global sensitivity analysis considers the complete input space.

4.1 Local Sensitivity Analysis

UncertaintyQuantification provides the computation of gradients for local, derivative based, sensitivity analysis through the `gradient` method. For a set of input variables and models the gradient can be calculated by passing a `DataFrame` of reference points to evaluate. The `gradient`

method will return a new `DataFrame` of the same size, containing the gradients. Internally, finite difference methods are used to approximate the gradients.

```
grads = gradient(model, inputs, reference, output)
```

Gradients can alternatively be estimated after transformation to the standard normal space. However, this is mainly provided in order to find the important direction required for line sampling [20].

```
grads = gradient_in_standard_normal_space(model, inputs, reference, output)
```

4.2 Global Sensitivity Analysis

For global sensitivity analysis [19] *UncertaintyQuantification* provides Sobol' indices, a form of variance based sensitivity analysis. This type of sensitivity analysis quantifies how much of the variance of a model output can be attributed to each of the inputs. Two main sensitivity measures are defined. For a model $Y = f(X_1, X_2, \ldots, X_n)$, the first order index is defined by

$$S_i = \frac{V_{X_i}(E_{\mathbf{X}_{\sim i}}(Y|X_i))}{V(Y)}, \tag{3}$$

where $\mathbf{X}_{\sim i}$ represents all input variables except i. The first order index captures the effect of varying X_i alone. The total effect index captures also the variance caused by all interactions of X_i with other variables. It is defined as

$$S_{T_i} = \frac{E_{\mathbf{X}_{\sim i}}(V_{X_i}(Y|\mathbf{X}_{\sim i}))}{V(Y)}. \tag{4}$$

To compute simulation based Sobol' indices we must first define the inputs and model as with all the previous examples. The indices can then be approximated by calling the `sobolindices` function. Quasi-Monte carlo methods like `SobolSampling` usually lead to better results than brute force Monte Carlo. The Sobol' indices for multiple models of the same inputs can be computed simultaneously by passing vectors of models and output symbols to the method.

```
# single model/output
indices = sobolindices(model, inputs, :y, SobolSampling(2000))
# multiple models/outputs
indices = sobolindices([m1, m2], inputs, [:y1, :y2], SobolSampling(2000))
```

The Sobol' indices are estimated according to Jansen (1999) [21] and Saltelli (2010) [22]. Bootstrapping is used to quantify the standard error of the estimators. For a complete example of how to compute Sobol' indices, see Section 6.2.

5 METAMODELIING

When dealing with complex, long running models, a single run of the model can take hours to days. In these cases, it is infeasible to run any analysis requiring a large number of samples. Quantities such as Sobol' indices for sensitivity analysis simply can not be computed. In these cases we can apply so called *metamodels* or *surrogate models*. These models are constructed from a small selection of evaluations of the true model and aim to approximate it as closely as possible. The metamodel itself is easy to evaluate and enables the analyses which where inaccessible for the original model.

The metamodels currently included in *UncertaintyQuantification* are the response surface methodology, polyharmonic splines and polynomial chaos expansion. Several designs of experiments (central composite design, Box-behnken design, factorial design), are provided to fit response surface models.

5.1 Polyharmonic spline

As a simple example we are going to construct a polyharmonic spline metamodel for a simple one-dimensional test function. A polyharmonic spline [23] is a combination of a low degree polynomial term and polyharmonic radial basis functions of the form

$$s(x) = \sum_{i=1}^{N} w_i \varphi(|\mathbf{x} - \mathbf{c}_i|) + \mathbf{v}^T \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix}. \tag{5}$$

The radial basis functions are $\varphi(r) = r^p, p = 1, 3, 5, \ldots$, and $\varphi(r) = r^p \ln(r), p = 2, 4, 6, \ldots$, with degree p and $r = |\mathbf{x} - c_i| = \sqrt{(\mathbf{x} - \mathbf{c}_i)^T (\mathbf{x} - \mathbf{c}_i)}$. The N center points to interpolate are denoted by $\mathbf{c}_i$.

The test function [24] is defined on $x \in [0, 1]$ as

$$f(x) = (6x - 2)^2 \sin(12x - 4). \tag{6}$$

To start, we set up the necessary input and the model. We sample 20 values of x using Sobol' sampling and evaluate the model for the sampled points. By using a quasi-Monte Carlo sampling scheme we make sure that the sampled points have adequate coverage of the input space.

```
x = RandomVariable(Uniform(0, 1), :x)

m = Model(df -> (6 .* x .- 2) .^ 2 .* sin.(12 .* x .- 4), :y)

data = sample(x, SobolSampling(20))
evaluate!(m, data)
```

In order to construct the polyharmonic spline (and determine the required weights $\mathbf{w}$ and $\mathbf{v}$), the sampled points are passed to the constructor along with the desired degree $p = 2$ and the `Symbol` of the output to fit the metamodel to.

```
ps = PolyharmonicSpline(data, 2, :y)
```

A plot comparing the metamodel to the original function and the sampled points can be seen in Figure 2. With just 20 sampled points, the spline is able to accurately approximate the test function.

6 NUMERICAL EXAMPLES

This section presents a few more complex examples than the ones presented in the previous sections.

6.1 Estimation of small failure probabilities

Subset simulation [see 10] is an advanced Monte Carlo simulation technique for the estimation of small failure probabilities. Here we solve a simple problem [taken from 25, page

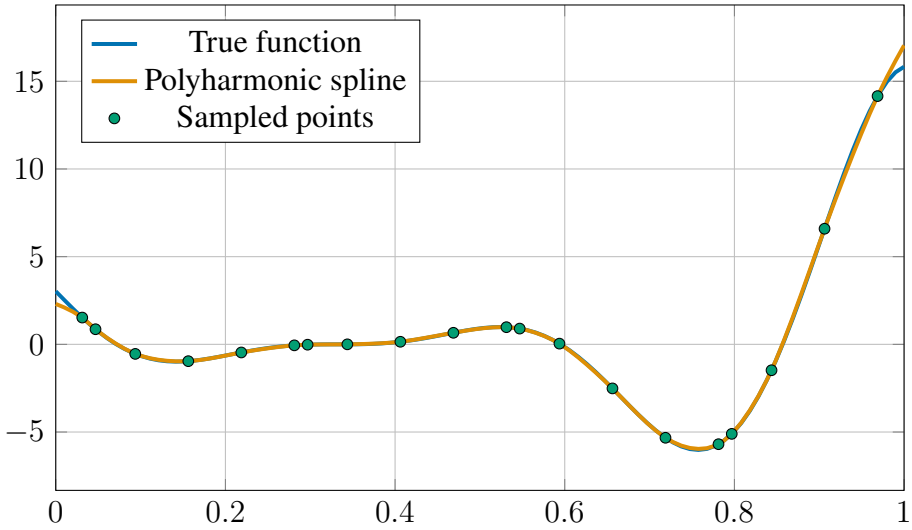


Figure 2: Polyharmonic spline metamodel compared to the true function.

9] where the response y depends on two independent random variables x_1 and x_2 following a standard normal distribution. The simple linear model is defined by

$$y(x_1, x_2) = x_1 + x_2 \tag{7}$$

with the failure domain

$$F = \{(x_1, x_2) : x_1 + x_2 > 9\}. \tag{8}$$

The analytical probability of failure can be calculated as

$$p_f = 1 - \Phi(\frac{9}{\sqrt{2}}) \approx 1 \times 10^{-10}, \tag{9}$$

where Φ is the standard normal cumulative densitiy function.

In order to solve this problem, we start by creating the two standard normal random variables and group them in a vector `inputs`.

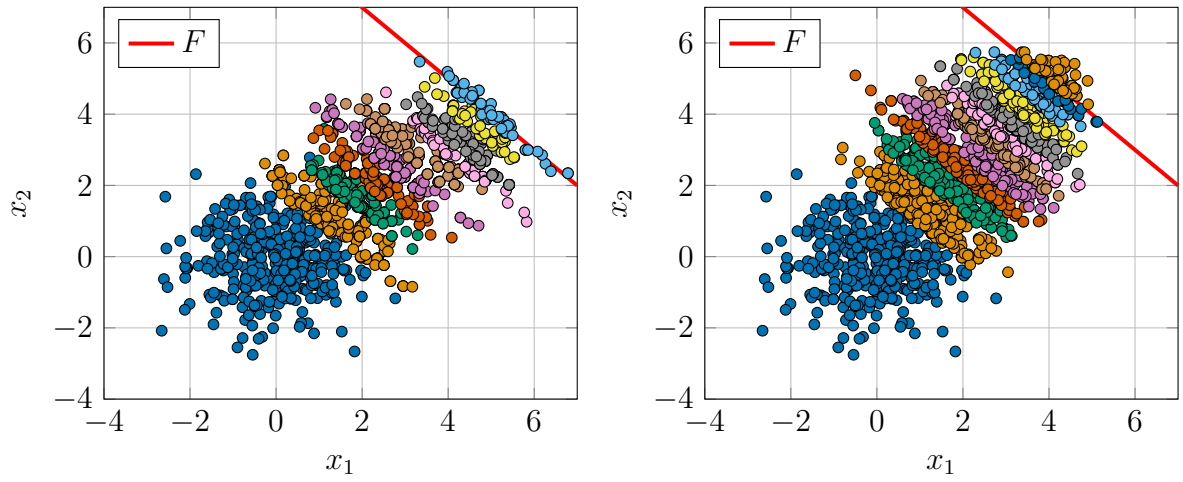
```
x1 = RandomVariable(Normal(), :x1)
x2 = RandomVariable(Normal(), :x2)
inputs = [x1, x2]
```

Next we define the model as

```
y = Model(df -> df.x1 + df.x2, :y)
```

where the first input is our function (which must accept a `DataFrame`) and the second the `Symbol` for the output variable.

To estimate a failure probability we need a performance function. This function, which accepts a `DataFrame` similar to the `Model`, must return a vector which is negative if a failure occurs for a sample. Here, a failure occurs if y exceeds 9.



(a) SubSetSimulation (b) SubSet-∞

Figure 3: Samples at the different levels for the two Subset simulations. Both simulations have been started from the same seed samples at the first level.

```
function g(df::DataFrame)
  return 9 .- df.y
end
```

Finally, we create the `SubSetSimulation` object and compute the probability of failure using a standard Gaussian proposal PDF. The value for the target probability of failure at each intermediate level is set to 0.1.

```
subset = SubSetSimulation(1000, 0.1, 10, Normal())
pf, cov, samples = probability_of_failure(y, g, inputs, subset)
```

Alternatively, instead of using the standard Subset simulation algorithm (which internally uses Markov Chain Monte Carlo), we can use `SubSetInfinity` to compute the probability of failure, see [26]. Here we use a standard deviation of 0.5 to create the proposal samples for the next level.

```
subset = SubSetInfinity(1000, 0.1, 10, 0.5)
pf, cov, samples = probability_of_failure(y, g, inputs, subset)
```

Figure 3 shows the samples at each level of the two Subset simulations. Both methods are able to estimate the small probability of failure with the MCMC based method yielding $p_f \approx 5.3912e-10$ and Subset-∞ $p_f \approx 3.13212e-11$. The plots show, that Subset-∞ required two extra levels to approximate the probability of failure. However, due to the numerical demand of the Markov chains, Subset-∞ runs about twice as fast for this example.

6.2 Global Sensitivity Analysis of the Ishigami function

The Ishigami function [27] given by

$$f(x) = \sin(x_1) + a \sin(x_2)^2 + b x_3^4 \sin(x_1), \tag{10}$$

with $x_i \sim U(-\pi, \pi)$ for $i = 1, 2, 3$, is often used to benchmark global sensitivity analysis methods. It shows strong nonlinearity and has an interesting dependence on x_3. The parameter

Table 1: Approximated first order and total effect Sobol' indices of the Ishigami function with estimated standard errors. The analytical solutions S_i and S_{T_i} are included for comparison.

x_i	$\hat{S}_i$	$\sigma_{\hat{S}_i}$	S_i	$\hat{S_{T_i}}$	$\sigma_{\hat{S_{T_i}}}$	S_{T_i}
x_1	0.291419	0.00964169	0.3138	0.536946	0.0129737	0.5574
x_2	0.436318	0.0102289	0.4424	0.439057	0.0130688	0.4424
x_3	-0.004927	0.0101326	0.0	0.243939	0.0130147	0.2436

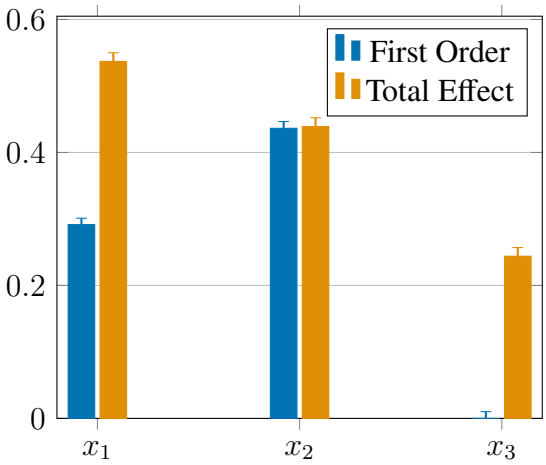


Figure 4: First order and total effect Sobol' indices of the Ishigami function.

a is typically set to 7 while b is either 0.1 [28] or 0.05 [29].

The following script will estimate the first order and total effect Sobol' indices of the Ishigami function using 10000 samples drawn from the Sobol' sequence. Using a quasi-Monte Carlo method instead of the brute force method improves convergence of the Sobol' indices.

```
x = RandomVariable.(Uniform(-pi,pi), [:x1, :x2, :x3])

a = Parameter(7, :a)
b = Parameter(0.1, :b)

function ishigami(df)
  return @. sin(df.x1) + df.a * sin(df.x2)^2 + df.b * df.x3^4 * sin(df.x1)
end

m = Model(ishigami, :y)

sobol = sobolindices(m, [x..., a, b], :y, SobolSampling(10000))
```

The Sobol' indices computed for the Ishigami function using 10000 samples are shown in Figure 4 and Table 1. S_3 being slightly negative can be attributed to numerical inaccuracies. Note also the interesting dependence on x_3. While S_3 suggests no influence on the function at all, S_{T_3} proves that x_3 has significant influence through interactions.

Computing thousands of model evaluations will be impossible for a complex model. To work around this problem, we can build an accurate metamodel of our model and use it to compute the Sobol' indices as seen in the next section.

Table 2: First order and total effect Sobol' indices of the Ishigami function estimated from the polynomial chaos expansion.

	$\hat{S}_i$	$\hat{S_{T_i}}$
x_1	0.312994	0.555787
x_2	0.444213	0.444213
x_3	5.10399e-32	0.242794

6.3 Metamodelling and Global Sensitivity Analysis

It is entirely possible to build a metamodel such as a response surface or polyharmonic spline based on a few model evaluations and use it to compute the Sobol' indices similarly to the previous section. However, a unique property of the polynomial chaos expansion (PCE) [30] is that it provides the Sobol' indices as a byproduct as they can be calculated from the coefficients of the PCE. Consider again the Ishigami function. In order to build a PCE metamodel we must first define the polynomial basis to be used.

We begin by selecting the degree $p = 8$ and the Legendre polynomials as a basis for the three input variables. Then we assemble the `PolynomialChaosBasis` from the combinations of degree up to p of our basis functions.

```
p = 8
bases = [LegendreBasis(), LegendreBasis(), LegendreBasis()]
psi = PolynomialChaosBasis(bases, p)
```

Internally, *UncertaintyQuantification* will perform an isoprobabilistic transformation of the inputs to the support of the basis functions (in this case $[0, 1]$ for all inputs). Next, the PCE metamodel is built using Gauss quadrature and the Sobol' indices are computed directly from the polynomial chaos expasion.

```
pce, samples = polynomialchaos(inputs, m, psi, :y, GaussQuadrature())
sobol = sobolindices(pce)
```

The resulting indices are presented in Table 2. No errors are estimated when computing the Sobol' indices from a polynomial chaos expansion. With just 729 model evaluations for $p = 8$ one can see that the results are much closer to the analytical solution than using 10000 samples with the true model. Note, that the number of required quadrature nodes grows quickly with increasing degree and the number of inputs. This issue can be eleviated by applying sparse quadrature grids. However, the implementation of sparse quadrature and other methods of estimating the PCE coefficients in *UncertaintyQuantification* is still ongoing.

7 CONCLUSION

This paper presented a new framework for uncertainty quantification called *UncertaintyQuantification.jl*. The framework has been developed as a module in the Julia programming language making use of its modern features. This work serves as an introduction to the framework. We presented its basic usage and outlined how to perform a few key analyses including reliability analysis, sensitivity analysis and metamodelling.

Ongoing development of the framework is focused on improving existing methods such as extending the polynomial chaos expansion with sparse quadratures, as well as adding more algo-

rithms. Features currently in development include more metamodels like Kriging and artificial neural networks, global sensitivity analysis with dependent inputs and the propagation of imprecise probabilities through probability boxes.

The source code of the framework is released under the MIT license and publicly available on Github. Contributions from other researchers are very welcome.

REFERENCES

[1] B. M. Adams, W. J. Bohnhoff, K. R. Dalbey, *et al.*, "Dakota, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis: Version 6.13 User's Manual," Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), SAND2020-12495, 2020. DOI: 10.2172/1817318.

[2] M. Baudin, A. Dutfoy, B. Iooss, and A.-L. Popelin, "Openturns: An industrial software for uncertainty quantification in simulation," in *Handbook of uncertainty quantification*, Springer, 2017, pp. 2001–2038. DOI: 10.48550/arXiv.1501.05242.

[3] E. Patelli, M. Broggi, M. d. Angelis, and M. Beer, "Opencossan: An efficient open tool for dealing with epistemic and aleatory uncertainties," in *Vulnerability, Uncertainty, and Risk: Quantification, Mitigation, and Management*, 2014, pp. 2564–2573. DOI: 10.1061/9780784413609.258.

[4] S. Marelli and B. Sudret, "UQLab: A Framework for Uncertainty Quantification in Matlab," in *Vulnerability, Uncertainty, and Risk*, American Society of Civil Engineers, 2014, pp. 2554–2563. DOI: 10.1061/9780784413609.257.

[5] A. Olivier, D. G. Giovanis, B. S. Aakash, M. Chauhan, L. Vandanapu, and M. D. Shields, "UQpy: A general purpose Python package and development environment for uncertainty quantification," *Journal of Computational Science*, vol. 47, p. 101 204, 2020, ISSN: 1877-7503. DOI: 10.1016/j.jocs.2020.101204.

[6] J. Behrensdorf, A. Gray, G. Agarwal, A. Perin, M. Luttmann, and L. Knipper, *FriesischScott/UncertaintyQuantification.jl: v0.6.1*, 2023. DOI: 10.5281/zenodo.3993816.

[7] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A Fresh Approach to Numerical Computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017, ISSN: 0036-1445. DOI: 10.1137/141000671.

[8] M. Besançon, T. Papamarkou, D. Anthoff, *et al.*, "Distributions.jl: Definition and modeling of probability distributions in the juliastats ecosystem," *Journal of Statistical Software*, vol. 98, no. 16, pp. 1–30, 2021, ISSN: 1548-7660. DOI: 10.18637/jss.v098.i16.

[9] P. S. Koutsourelakis, H. J. Pradlwarter, and G. I. Schuëller, "Reliability of structures in high dimensions, part I: Algorithms and applications," *Probabilistic Engineering Mechanics*, vol. 19, no. 4, pp. 409–417, 2004, ISSN: 0266-8920. DOI: 10.1016/j.probengmech.2004.05.001.

[10] S.-K. Au and J. L. Beck, "Estimation of small failure probabilities in high dimensions by subset simulation," *Probabilistic Engineering Mechanics*, vol. 16, no. 4, pp. 263–277, 2001, ISSN: 02668920. DOI: 10.1016/S0266-8920(01)00019-4.

[11] M. Sklar, "Fonctions de repartition a n dimensions et leurs marges," *Publ. inst. statist. univ. Paris*, vol. 8, pp. 229–231, 1959.

[12] H. Joe, *Dependence Modeling with Copulas*. 2015, ISBN: 978-1-4665-8322-1.

[13] A. Gray, J. Behrensdorf, amrods, and C. Schilling, *AnderGray/BivariateCopulas.jl: 0.1.4*, 2023. DOI: 10.5281/zenodo.7822923.

[14] *JuliaParallel/ClusterManagers.jl: v0.4.4*, 2023. [Online]. Available: https://github.com/JuliaParallel/ClusterManagers.jl.

[15] *jverzani/Mustache.jl: v1.0.14*, 2022. [Online]. Available: https://github.com/jverzani/Mustache.jl.

[16] *JuliaIO/Formatting.jl: v0.4.2*, 2020. [Online]. Available: https://github.com/JuliaIO/Formatting.jl.

[17] A. M. Hasofer and N. C. Lind, "Exact and Invariant Second-Moment Code Format," *Journal of the Engineering Mechanics Division*, vol. 100, no. 1, pp. 111–121, 1974. DOI: 10.1061/JMCEA3.0001848.

[18] R. Rackwitz and B. Fiessler, "Structural Reliability under combined random load sequences," *Computers & Structures*, vol. 9, no. 5, pp. 489–494, 1978, ISSN: 0045-7949. DOI: 10.1016/0045-7949(78)90046-9.

[19] A. Saltelli, M. Ratto, T. Andres, *et al.*, *Global Sensitivity Analysis. The Primer*. Chichester, UK: John Wiley & Sons, Ltd, 2007. DOI: 10.1002/9780470725184.

[20] H. J. Pradlwarter, G. I. Schuëller, P. S. Koutsourelakis, and D. C. Charmpis, "Application of line sampling simulation method to reliability benchmark problems," *Structural Safety*, A Benchmark Study on Reliability in High Dimensions, vol. 29, no. 3, pp. 208–221, 2007, ISSN: 0167-4730. DOI: 10.1016/j.strusafe.2006.07.009.

[21] M. J. W. Jansen, "Analysis of variance designs for model output," *Computer Physics Communications*, vol. 117, no. 1, pp. 35–43, 1999, ISSN: 0010-4655. DOI: 10.1016/S0010-4655(98)00154-4.

[22] A. Saltelli, P. Annoni, I. Azzini, F. Campolongo, M. Ratto, and S. Tarantola, "Variance based sensitivity analysis of model output. Design and estimator for the total sensitivity index," *Computer Physics Communications*, vol. 181, no. 2, pp. 259–270, 2010, ISSN: 00104655. DOI: 10.1016/j.cpc.2009.09.018.

[23] R. K. Beatson, M. J. D. Powell, and A. M. Tan, "Fast evaluation of polyharmonic splines in three dimensions," *IMA Journal of Numerical Analysis*, vol. 27, no. 3, pp. 427–450, 2007, ISSN: 0272-4979. DOI: 10.1093/imanum/drl027.

[24] Alexander I.J. Forrester and Andras Sobester and Andy J. Keane, *Engineering Design via Surrogate Modelling: A Practical Guide*. Wiley, 2008.

[25] K. Zuev, "Subset Simulation Method for Rare Event Estimation: An Introduction," *arXiv: 1505.03506 [stat]*, 2015. DOI: 10.48550/arXiv.1505.03506.

[26] E. Patelli and S. K. Au, "Efficient Monte Carlo algorithm for rare failure event simulation," in *12th International Conference on Applications of Statistics and Probability in Civil Engineering, ICASP 2012*, CAN: University of British Columbia, 2015, ISBN: 978-0-88865-245-4.

[27] T. Ishigami and T. Homma, "An importance quantification technique in uncertainty analysis for computer models," in *Proceedings. First International Symposium on Uncertainty Modeling and Analysis*, 1990, pp. 398–403. DOI: 10.1109/ISUMA.1990.151285.

[28] A. Marrel, B. Iooss, B. Laurent, and O. Roustant, *Calculations of sobol indices for the gaussian process metamodel*, 2009. DOI: 10.1016/j.ress.2008.07.008.

[29] I. M. Sobol' and Y. L. Levitan, "On the use of variance reducing multipliers in Monte Carlo computations of a global sensitivity index," *Computer Physics Communications*, vol. 117, no. 1, pp. 52–61, 1999, ISSN: 0010-4655. DOI: 10.1016/S0010-4655(98)00156-8.

[30] B. Sudret, "Global sensitivity analysis using polynomial chaos expansions," *Reliability Engineering & System Safety*, vol. 93, no. 7, pp. 964–979, 2008, ISSN: 09518320. DOI: 10.1016/j.ress.2007.04.002.