`UQpy` - Uncertainty Quantification with Python

Authors: Michael D. Shields[*], Dimitris G. Giovanis[†]

Contributors: Aakash Bangalore-Satish, Mohit Chauhan, Audrey Olivier, Lohit Vandanapu, Jiaxin Zhang

Shields Uncertainty Research Group (SURG)

Johns Hopkins University, USA

Version 2.0.0

[*] `michael.shields@jhu.edu`

[†] `dgiovan1@jhu.edu`

Contents

1 Overview

UQpy (Uncertainty Quantification with Python) is a general purpose Python toolbox for modeling uncertainty in the simulation of physical and mathematical systems. The code is organized as a set of modules centered around core capabilities in Uncertainty Quantification (UQ) as illustrated in Figure 1. The modules are distinct, but are designed to be easily extensible (new capabilities can be easily added and integrated into the code, see Section 7) and to easily call one another.

The UQpy workflow is simple. Each module, as illustrated in Figure 1, contains a set of classes that perform various operations in UQ. A list of the current capabilities for each module is provided in Table 1. A list of expanded capabilities that are currently in development is provided in Table 2.

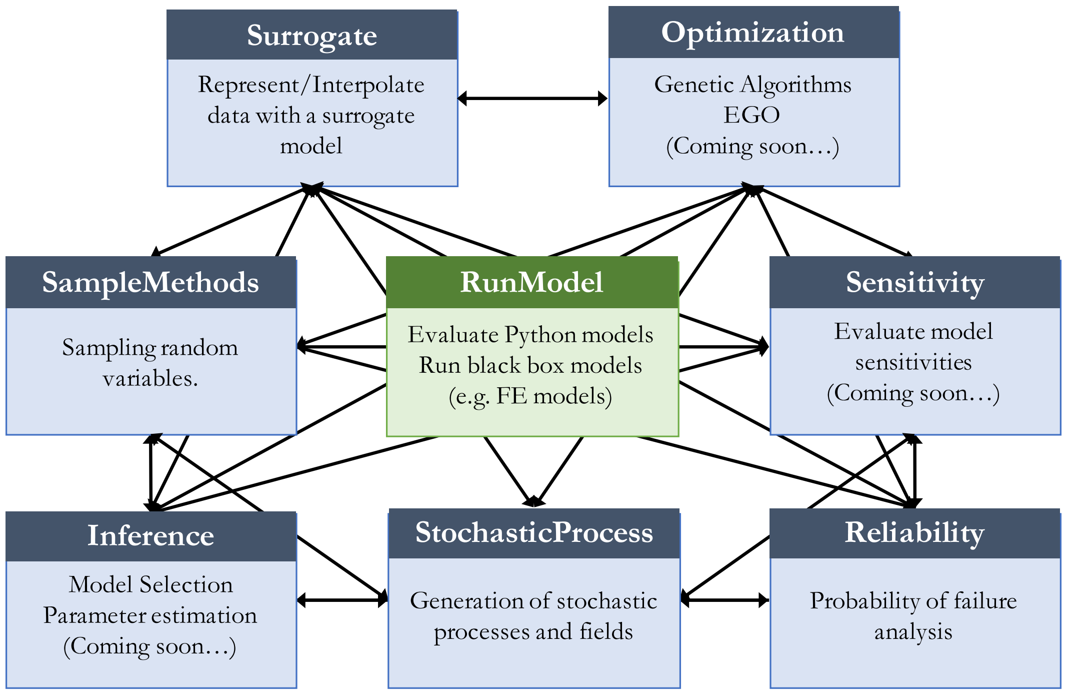


Figure 1: `UQpy` modules and their basic architecture.

Modules and Classes in `UQpy` are invoked using standard Python conventions. Because each module is organized into a set of classes, it is straightforward to add a new capability to `UQpy` by simply writing a new class into the appropriate module (although some care should be taken to ensure consistency in input/output naming and data type conventions). Moreover, because of its module-class structure, the various classes can easily invoke one-another and can be combined in any way the user desires. A simple example of this is that the `SubsetSimulation` class in the `Reliability` module invokes the `MCMC` class from the `SampleMethods` module.

The various classes and modules interface in a straightforward manner with computational models of physical or mathematical systems through the `RunModel` module shown in the center of the chart in Figure 1. The `RunModel` module allows `UQpy` to serve not just as a useful tool for performing UQ operations, but also as the driver for a complete uncertainty study - including preprocessing operations, submission and execution of computational model evaluations, and monitoring and post-processing of results. Thus, it is amenable to performing adaptivity UQ anayses. The `RunModel` module, detailed in Section 5.1, is designed to interface with any user-defined third-party computational

Table 1: Current `UQpy` capabilities organized by Module and Class structure.

Module	**Class**	**Description**	**Version**
`RunModel`	`RunModel`	Execute computational model	1.0.0
`Distributions`	`Distribution`	Define a Distribution object in `UQpy`	2.0.0
	`Marginals`	C	2.0.0
	`Copula`	Defines dependence models for distributions	2.0.0
`SampleMethods`	`MCS`	Monte Carlo Sampling	1.1.0
	`LHS`	Latin Hypercube Sampling	1.1.0
	`STS`	Stratified Sampling	1.1.0
	`MCMC`	Markov Chain Monte Carlo	1.1.0
	`IS`	Importance Sampling	1.3.0
	`RSS`	Refined Stratified Sampling	2.0.0
	`Simplex`	Uniform Sampling over a simplex element	2.0.0
`Transformations`	`Correlate`	Induces correlation	1.1.0
	`Decorrelate`	Removes correlation	1.1.0
	`Nataf`	Nataf transformation	1.1.0
	`InvNataf`	Inverse Nataf transformation	1.1.0
`Surrogates`	`SROM`	Stochastic Reduced Order Model	1.0.0
	`Kriging`	Gaussian Process Regression (Kriging)	2.0.0
`Reliability`	`SubsetSimulation`	Subset Simulation	1.0.0
	`TaylorSeries`	First Order Reliability Method (FORM) Second Order Reliability Method (SORM)	2.0.0
`Inference`	`InfoModelSelection`	Information Theoretic Model Selection (AIC/BIC)	2.0.0
	`BayesModelSelection`	Bayesian Model Selection	2.0.0
	`MLEstimation`	Maximum Likelihood Parameter Estimation	2.0.0
	`BayesParameterEstimation`	Bayesian Parameter Estimation	2.0.0
	`Model`	Model Definition for Model Selection	2.0.0
`StochasticProcess`	`SRM`	Spectral Representation Method	2.0.0
	`BSRM`	Bispectral Representation Method	2.0.0
	`KLE`	Karhunen-Loéve Expansion	2.0.0
	`Translation`	Translation Process	2.0.0
	`ITAM`	Iterative Translation Approximation Method	2.0.0
`Utilities`	`Diagnostics`	Diagnostic tools for `UQpy` objects	2.0.0

model (through Python scripts) or directly with a Python model.

Table 2: Future UQpy capabilities organized by Module and Class structure.

Module	**Class**	**Description**	**Version**
SampleMethods	LSS	Latinized Stratified Sampling	3.0.0
	PSS	Partially Stratified Sampling	3.0.0
	LPSS	Latinized Partially Stratified Sampling	3.0.0
	LRSS	Latinized Refined Stratified Sampling	3.0.0
	SparseGrid	Sparse Grid Cubature Sampling	3.0.0
	QMC	Quasi Monte Carlo	3.0.0
	HMC	Hamiltonian Monte Carlo	3.0.0
	Composition	Composition Sampling Method	3.0.0
	ASGC	Adaptive Sparse Grid Collocation	3.0.0
	SCAMR	Stochastic Collocation with Adaptive Mesh Refinement	3.0.0
Surrogates	PCE	Polynomial Chaos Surrogate	3.0.0
	ANN	Artificial Neural Network Surrogate	3.0.0
	SSC	Simplex Stochastic Collocation	3.0.0
	VSSC	Variance-based Simplex Stochastic Collocation	3.0.0
	Grassmann	Grassmann Manifold Projection Surrogate	3.0.0
Reliability	TRS	Targeted Random Sampling	3.0.0
	SESS	Surrogate Enhance Stochastic Search	3.0.0
	AK-MCS	Adaptive Kriging Monte Carlo Simulation	3.0.0
Inference	KDE	Kernel Density Estimation	3.0.0
Optimization	EGO	Efficient Global Optimization	3.0.0
	GA	Genetic Algorithms	3.0.0
Sensitivity	Sobol	Sobol Indices	3.0.0
	PCESobol	Polynomial Chaos Sobol Indices	3.0.0
DimensionReduction	POD	Proper Orthogonal Decomposition	3.0.0
	DiffMap	Diffusion Maps	3.0.0

2 Installing UQpy

UQpy is written in the Python 3 programming language and requires a Python interpreter 3.6+ installed on your computer. UQpy is distributed through the Python Package Index, PyPI, and can be installed using a simple pip command on the terminal as follows:

```
pip install UQpy
```

Upon installation, the UQpy software modules are installed in the site-packages directory of the user's Python installation. For example, within the user's Python (version 3.6) installation, the installed modules can be found at:

```
./lib/python3.6/site-packages/UQpy
```

`UQpy` can be uninstalled in a similar manner using pip:

```
pip uninstall UQpy
```

2.1 Manual Installation

Alternatively, `UQpy` can be installed from GitHub directly by typing the following commands in the terminal:

```
git clone https://github.com/SURGroup/UQpy.git

cd UQpy/

python setup.py install
```

Direct installation from GitHub is equivalent to pip installation.

`UQpy` can be uninstalled using pip as:

```
pip uninstall UQpy
```

2.2 Developer Installation

Users interested in developing new capabilities in `UQpy` may install it as a developer. This is achieved by typing the following commands in the terminal:

```
git clone https://github.com/SURGroup/UQpy.git

cd UQpy/

python setup.py develop
```

Installing as a developer allows the user to maintain a local copy of `UQpy` (located in a directory of the user's choosing) that can be edited – with changes being recognized by the `UQpy` "installation". Installing as a developer does not install the software directly to site-packages as in the installation procedures above. Instead, developer installation creates an 'egg-link' (`UQpy.egg-link`) in the site-packages that directs `UQpy` calls to the user's local, editable copy of the software. For more details, see the following link:

`http://setuptools.readthedocs.io/en/latest/setuptools.html#development-mode`

3 License

UQpy is distributed under the MIT license.

4 `UQpy` Modules, Classes, & Functions

`UQpy` is currently structured according to eight core modules (see Figure 1), each centered around specific functionalities, plus a `Utilities` module that provides support tools for the core modules. Two additional core modules are currently under development. The complete list of modules are as follows:

Core Modules

1. `RunModel`: This module contains the `RunModel` class that allows `UQpy` to initiate simulations using Python or third-party computational solvers, and monitor and post-process simulation results. See Section 5.1.

2. `SampleMethods`: This module contains a set of classes to draw samples

from random variables. These samples may be randomly drawn, as in Monte Carlo sampling, or they may be deterministically drawn as in sparse-grid or quasi-Monte Carlo sampling. The module also contains a number of variance reduction techniques.

3. `Inference`: This module contains a set of classes and functions to conduct probabilistic inference. The module contains methods that are based on Bayesian, frequentist, likelihood, and information theories.

4. `Reliability`: This module contains a set of classes to estimate rare event probabilities and probability of failure.

5. `Surrogate`: This module contains a set of classes for building surrogate models, meta-models, or emulators.

6. `StochasticProcess`: This module contains a set of classes and functions for simulation of stochastic processes and fields.

7. `Transformations`: This module contains a set of classes for isoprobabilistic transformations.

8. `Sensitivity`: (Coming in Version 3.0.0) This module will contain a set of classes for performing global and local sensitivity analysis.

9. `Optimization`: (Coming in Version 3.0.0) This module will contain a set of classes to perform optimization for stochastic problems.

Support Modules

1. `Distributions`: This module contains a set of classes for defining probability distribution objects in `UQpy`. It contains several supported distributions and associated functions (e.g. pdf, cdf, moments, random numbers, fit, inverse cdf, log_pdf) as well as allowing the user to define custom distributions.

2. `Utilities`: This module contains a set of classes and functions that are used in support of the other modules.

The following sections detail the classes and functions in each module with reference to examples that illustrate their use.

5 Core Modules

5.1 `RunModel` Module

The `RunModel` module is at the heart of `UPQpy`. It is a powerful module which enables `UQpy` to drive probabilistic computational modeling. This module can interact with and call third-party software, which allows batch processing. Using the `RunModel` module only requires familiarity with Python programming language and the domain-specific knowledge of the model being evaluated. The `RunModel` module allows parallel computing such that, when processing multiple jobs, the jobs can be distributed over multiple processes or threads. In the case of cluster computing, where the jobs are performed over multiple cores on multiple compute nodes, `RunModel` is powered by GNU parallelization (see Section 5.1.5). For parallelization across a single compute node or workstation, `RunModel` employs the Python `concurrent` package when run in combination with a Python computational model, and GNU parallel when running a third-party software model.

5.1.1 `RunModel` Workflows

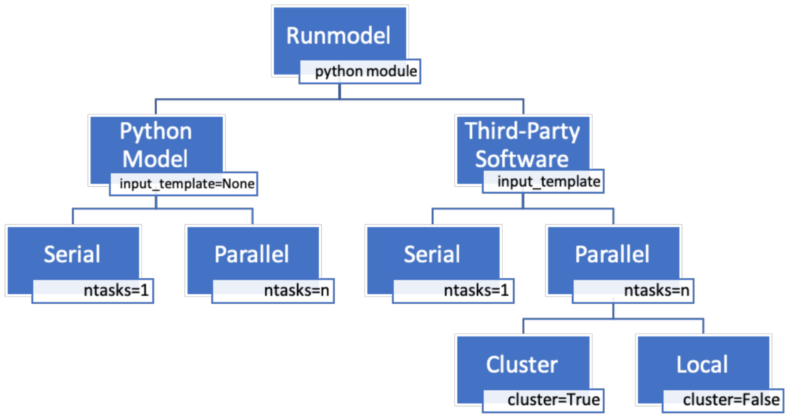


Figure 2: `RunModel` workflows and variables which trigger the different workflows.

`RunModel` class has four basic workflows delineated in two levels. At the first level, `RunModel` can be used in combination with either a Python computational model, in which case the model is imported and run directly, or in combination with a third-party software model. When running with a third-party software model, `RunModel` interfaces with the model through text-based input files and serves as the "driver" to initiate the necessary calculations. At the second level, the jobs that are run by `RunModel` can either be executed in series or in parallel. Within the third-party model parallel execution workflow, there are two cases, which are triggered by the `cluster` variable. In the following sections we will discuss the workflows in detail.

5.1.2 `UQpy.RunModel.RunModel`

The `RunModel` module consists of a single class, also called `RunModel`, that can be imported using the following command:

```
from UQpy.RunModel import RunModel
```

The minimum required and optional attributes of the `RunModel` class depend on the desired workflow and are listed below.

For execution of a Python model:

RunModel Class Attribute Definitions: Python model workflow			
Attribute	**Input/Output**	**Required**	**Optional**
samples	Input	⋆	
model_script	Input	⋆	
model_object_name	Input		⋆
ntasks	Input		⋆
verbose	Input		⋆
model_dir	Input		⋆
qoi_list	Output		

For execution of a third-party software model:

`RunModel` Class Attribute Definitions: Third-party model workflow			
Attribute	**Input/Output**	**Required**	**Optional**
`samples`	Input	⋆	
`model_script`	Input	⋆	
`input_template`	Input	⋆	
`var_names`	Input		⋆
`output_script`	Input		⋆
`output_object_name`	Input		⋆
`ntasks`	Input		⋆
`cores_per_task`	Input		⋆
`nodes`	Input		⋆
`resume`	Input		⋆
`verbose`	Input		⋆
`model_dir`	Input		⋆
`cluster`	Input		⋆
`qoi_list`	Output		

A brief description of each attribute can be found in the table below:

`RunModel` Class Attributes			
Attribute	**Type**	**Options**	**Default**
`samples`	*list* or *ndarray*		`None`
`model_script`	*string*		`None`
`model_object_name`	*string*		`None`
`input_template`	*string*		`None`
`var_names`	*list*		`None`
`output_script`	*string*		`None`
`output_object_name`	*string*		`None`
`ntasks`	*integer*		`1`
`cores_per_task`	*integer*		`1`
`nodes`	*integer*		`1`
`resume`	*bool*		`False`
`verbose`	*bool*		`False`
`model_dir`	*str*		`None`
`cluster`	*bool*		`False`

Detailed Description of `RunModel` Class Attributes:

Input Attributes:

- `samples`:
  Samples to be passed as inputs to the model. Samples can be passed either as an *ndarray* or a *list*.

  If an *ndarray* is passed, each row of the *ndarray* contains one set of samples required for one execution of the model. (The first dimension of the *ndarray* is considered to be the number of rows.)

  If a *list* is passed, each item of the *list* contains one set of samples required for one execution of the model.

- `model_script`
  The filename (with extension) of the Python script which contains commands to execute the model. The model script must be present in the current working directory from which `RunModel` is called.

  The model script is used in different ways for the Python and third-party software workflows. For further details, see Section 5.1.8.

- `model_object_name`
  In the Python model workflow, `model_object_name` specifies the name of the function or class within `model_script` that executes the model. If there is only one function or class in the `model_script`, then it is not necessary to specify `model_object_name`. If there are multiple objects within the `model_script`, then `model_object_name` must be specified.

  `model_object_name` is only used with the Python model workflow, which imports the model object into the working Python environment. When running a third-party software model, `RunModel` calls the `model_script` from the command line and passes an input (i.e., the sample number) to the `model_object`. Several approaches are possible to facilitate calling the `model_script` and passing an input to the `model_object`. Refer Section 5.1.5 for an illustration using the module `Fire` to do this.

- `input_template`:
  The name of the template input file which will be used to generate input files for each run of a third-party model.

  When operating `RunModel` with a third-party software model, `input_template` must be specified. For details on setting up template input files, see Section 5.1.8.

  `input_template` is not used in the Python model workflow.

- `var_names`:
  A list of strings containing the names of the variables present in the template input file specified by `input_template`.

  If an `input_template` is provided and a list of variable names is not passed, i.e. if `var_names = None`, then the default variable names *x0, x1, x2, ..., xn* are created and used by `RunModel`, where n is the number of variables. The number of variables is equal to the shape of the first row if `samples` is passed as an *ndarray* or the shape of the first item if `samples` is passed as a *list*.

  For additional details on how variable names are used in the template input file to generate run files, see Section 5.1.8.

  `var_names` is not used in the Python model workflow.

- `output_script`:
  The filename of the Python script which contains the commands to process the output from third-party software model evaluation. The `output_script` is used to return the output quantities of interest to `RunModel` for subsequent `UQpy` processing (e.g. for adaptive methods that utilize the results of previous simulations to initialize new simulations). See Section 5.1.8 for further details.

  `output_script` is not used in the Python model workflow. In the Python model workflow, all model postprocessing is handled within `model_script`. See Section 5.1.8 for further details.

  If, in the third-party software model workflow, `output_script = None` (the default), then `RunModel.qoi_list` is empty and postprocessing must be handled outside of `UQpy`.

- `output_object_name`:
  The name of the function or class that is used to collect the output values from third-party software model output files.

  If the object is a class named `cls`, for example, the quantity of interest extracted from the model output must be saved as `cls.qoi`. If it is a function, it should return the output quantity of interest. If there is only one function or only one class in `output_script`, then it is not necessary to specify `output_object_name`. If there are multiple objects in `output_script`, then `output_object_name` must be specified.

  `output_object_name` is not used in the Python model workflow.

- `ntasks`:
  Number of tasks to be run in parallel.

  By default, `ntasks = 1` and model evaluations are executed serially. Setting `ntasks` equal to a positive integer greater than 1 will trigger the parallel workflow.

  RunModel uses GNU parallel to execute models which require an input template in parallel and the `concurrent` module to execute Python models in parallel. Further details can be found in Sections 5.1.3 and 5.1.5.

- `cores_per_task`:
  Number of cores to be used by each task.

  In cases where a third-party model runs across multiple cores in a cluster, this optional attribute allocates the necessary resources to each model evaluation. `RunModel` does this by using the `SLURM` command `srun` in addition to `GNU parallel` and allocating `cores_per_task` number of cores per each execution of the model. When a third-party model is run in parallel on a machine which does not use `SLURM` workload manager, (typically, a laptop/personal computer), `GNU parallel` can only specify the number of jobs to be executed in parallel and not the number of cores to be used for each job.

  `cores_per_task` is not used in the Python model workflow.

- `nodes`:
  Number of nodes across which to distribute a single task on an HPC cluster in the third-party software model parallel workflow.

  If a task needs to be split across more than one compute node, `nodes` must be specified. For example, the Maryland Advanced Research Computing Center (MARCC), an HPC shared by Johns Hopkins University and the University of Maryland, a typical compute node has 24 cores and 128 GB of memory. If each task in the parallel job requires more resources than that available on a single compute node of the cluster, it is necessary to pass in a value for `nodes` which is greater than 1.

  `nodes` is passed as an argument to `SLURM`'s `srun` command and should only be changed by users familiar with the `srun`. Further details regarding the SLURM workload manager can be found here `https://slurm.schedmd.com`

  `nodes` is not used in the Python model workflow.

- `resume`:
  If `resume = True`, GNU parallel enables `UQpy` to resume execution of any model evaluations that failed to execute in the third-party software model workflow.

  To use this feature, execute the same call to `RunModel` which failed to complete but with `resume = True`. The same set of samples must be passed to resume processing from the last successful execution of the model.

  `resume` is not used in the Python model workflow.

- `verbose`:
  Set `verbose = True` if you want `RunModel` to print status messages to the terminal during execution. `verbose = False` by default.

- `model_dir`:
  Specifies the name of the sub-directory from which the model will be executed and within which output files will be saved.

  `model_dir = None` by default, which results in model execution from the Python current working directory. If `model_dir` is passed a string, then a new directory is created by `RunModel` within the current directory whose name is `model_dir` appended with a timestamp. See Section 5.1.7 and Figure 3 for more details.

- `cluster`:
  Set `cluster = True` if executing on an HPC cluster. Setting `cluster = True` enables `RunModel` to execute the model using the necessary SLURM commands. `cluster = False` by default.

  `RunModel` is configured for HPC clusters that operate with the SLURM scheduler. In order to execute a third-party model with `RunModel` on an HPC cluster, the HPC must support SLURM commands.

  `cluster` is not used for the Python model workflow.

Output Attributes:

- `qoi_list`:
  A list containing the output quantities of interest extracted from the model output files by `output_script`. This is a list of length equal to the number of simulations. Each item of this list contains the quantity of interest from the associated simulation.

5.1.3 `RunModel`: Python model workflow - serial execution

A common workflow in `UQpy` is when the computational model being evaluated is written in Python. This workflow invoked by calling `RunModel` without specifying an `input_template` (i.e. `input_template = None`) and setting `model_script` to the user-defined Python script containing the model. This python model is run serially by setting `ntasks = 1`.

`UQpy` imports the `model_script` and executes the object defined by `model_object_name`. The structure of the model object should be such that it should accept one sample as the input. If the model object is a Class, the quantity of interest must be stored as an attribute of the class `self.qoi`. If the model object is a function, it must return the quantity of interest after execution. In serial execution, the Python model is run with a different sample in every run.

Samples for how the Python model may be structured are provided below.
Example: Model object as a class:

```
class ModelClass:
    def __init__(self, input=one_sample):
        Execute the model using the input and get the output
        self.qoi = output
```

Exampel: Model object as a function:

```
def model_function(input=one_sample):
    Execute the model using the input and get the output
    return output
```

5.1.4 `RunModel`: Python model workflow - parallel execution

The python model is executed in parallel by setting `ntasks` equal to the desired number of tasks (greater than 1) to be executed concurrently. The model should be defined as explained in Section 5.1.3, i.e., in the same way as for the serial execution case. `RunModel` uses the python library `concurrent` for parallel execution of python models, which restricts parallelization to the cores available within a single node (if running on a cluster).

5.1.5 `RunModel`: Third-party software model workflow - serial execution

The `RunModel` class also supports running models using third-party software. This worrkflow uses a template input file (`input_template`) to pass information from `UQpy` to the third-party model, and a Python script to process the outputs and collect the results after post-processing.

This workflow operates in three steps as explained in the following.

Step 1:
`UQpy` takes the file `input_template` and generates an indexed set of input files, one for each set of sample values passed through the `samples` input. For example, if the name of the template input file is *input.inp*, then `UQpy` generates indexed input files by appending the sample number between the filename and extension, as *input_1.inp*, *input_2.inp*, ... , *input_n.inp*, where n is the number of sample sets in `samples`. The details of how the `input_template` should be structured are discussed in Section 5.1.8. During serial execution, one input file is generated, the model is executed, another input file is generated, the model is executed, and so on.

Step 2:
The third-party software model is executed for each set of sample values using the indexed model input file generated in Step 1 by calling the Python script specified in `model_script` and passing the sample index. This can be done either serially or in parallel over multiple processors (which may be performed over multiple nodes of an HPC cluster). For serial execution, we should set the parameter `ntasks = 1`.

Step 3:
For each simulation, the third-party model generates some set of outputs in Step 2. The user-defined `output_script` is used to post-process these outputs and return them to `RunModel` in a list form. This script should extract any desired quantity of interest from the generated output files, again using the sample index to link model outputs to their respective sample sets.

`UQpy` imports the `output_script` and executes the object defined by `output_object_name`. The structure of the output object must be such that it accepts, as input, the sample index. If the output object is a Class, the quantity of interest must be stored as an attribute of the class `self.qoi`. If the output object it is a function, it must return the quantity of interest after execution. More details specifying the structure of `output_script` and the associated output object can be found in Section 5.1.8.

Finally, because `UQpy` imports the `output_script` and executes it within `RunModel`, the values returned by the output object are directly stored according to their sample index in the `RunModel` attribute `qoi_list`.

5.1.6 `RunModel`: Third-party software model workflow - parallel execution

Parallel execution in RunModel module is carried out by the GNU parallel library [14]. GNU parallel is essential and must be installed on the computer running the model. Information regarding how to install GNU parallel is provided at `https://www.gnu.org/software/parallel`. For Mac users, a simple command

```
brew cask install parallel
```

can be used for installation. For Linux users,

```
sudo apt-get install parallel
```

should install the package. Parallel execution is actiavted in runModel workflow by setting the parameter `ntasks>1`. The key difference in therms of the workflow is listed below.

Step 1:
During parallel execution, all required input files are generated prior to model execution as opposed to serial execution where input files are generated individually for each run.

Step 2:
GNU parallel divides the total number of jobs into a number of chunks specified by the variable `ntasks`. `ntasks` number of jobs are executed in parallel and this continues until all the jobs finish executing. ote theat each job can be executed across multiple CPUs whe cluster=True using the SLURM workload manager. This is sepcified by setting cores_per_task and nodes appropriately, details can be seen in Section 5.1.2. Whether in serial or parallel, the sample index is used by `RunModel` to keep track of model execution and to link the samples to their corresponding outputs. `RunModel` achieves this by consistently naming all the input files using the sample index (see Step 1) and passing the sample index into `model_script`. More details on the precise structure of `model_script` are discussed in Section 5.1.8.

Step 3:
No key difference between the serial and parallel workflow in terms of output processing. Output processing in the paralle case is done after all the runs

are completed whereas in the serial case it is done after every run.

5.1.7 Directory structure during model evaluation

To execute `RunModel`, the directory from where `RunModel` is called must contain the necessary files (i.e., `model_script`, `input_template`, and `output_script`) along with any other files required for model evaluation. These may include, among other things, compiled executable files for third-party software that runs locally. There is an option to specify a `model_dir` as an input to `RunModel`. If a `model_dir` is specified, `RunModel` creates a new directory whose name is given by appending a timestamp corresponding to the time of executing the model to `model_dir`. All the files in the working directory are copied to the newly created model directory as illustrated in Figure 3 and this directory becomes the working directory for executing the model. If a `model_dir` is not specified, the current directory is the working directory for model execution.

To avoid cluttering the working directory with outputs, RunModel creates a directory for each execution of the model and saves the output generated during the model execution within the corresponding directory. `RunModel` generates the directory name for the sample as `run_n_timestamp`, where `n` goes from `0` to `number of samples-1`, and `timestamp` corresponds to the time at the beginning of the first simulation of the parallel job. See Figure 4 for an illustration.

Within the directory for each run, `RunModel` creates a new directory `InputFiles` and deposits the input files generated in Step 1 above into this directory. The user's model script must retrieve the relevant input file during the model execution. During model execution, `RunModel` first copies all the files in the working directory to the directory for each sample, executes the model, and then deletes all the files copied into this directory from the working directory. Any output generated either during model execution or during output processing remains in this directory along with the `InputFiles` directory. See Figure 5 for an illustration.

5.1.8 Files and scripts used by `RunModel`

As discussed in the sections above and illustrated in the examples, the `RunModel` class utilizes a python script to execute the computational model (`model_script`), a python script to extract the output (`output_script`) and

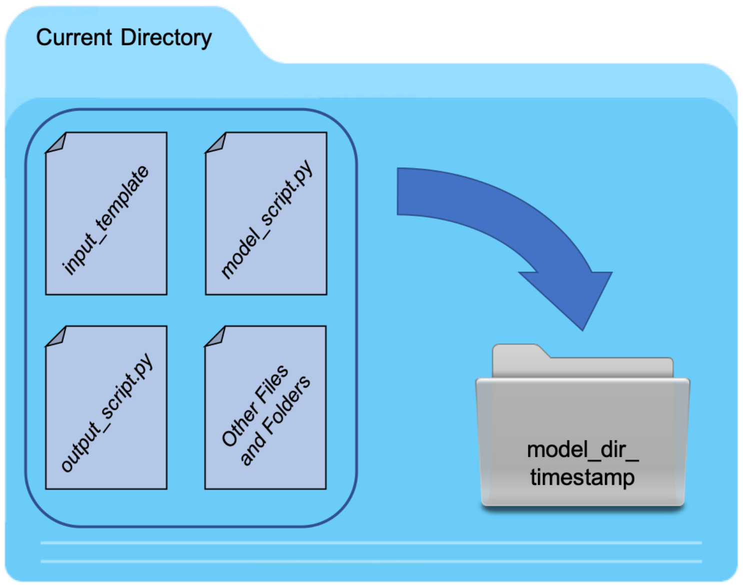


Figure 3: If a `model_dir` is specified, `RunModel` first copies all files into a subdirectory of the working directory called `model_dir_timestamp` where all computations will be performed and this directory becomes the working directory. If `model_dir` is not specified, the current directory is the working directory.

a template input file (`input_template`). This section is intended to provide a closer look at each of these files, their structure, and when/if they are required.

`input_template`:

- `input_template` is a user-defined file that is is used only when executing a third-party software model with `RunModel`. As the name implies, `input_template` serves as a template of the model input file from which individual model input files will be generated for each model evaluation. The model input file is typically an ASCII text-based file that defines all parameters, geometry, material, properties, etc. of the computational model. For each individual model evaluation, `RunModel` will modify this

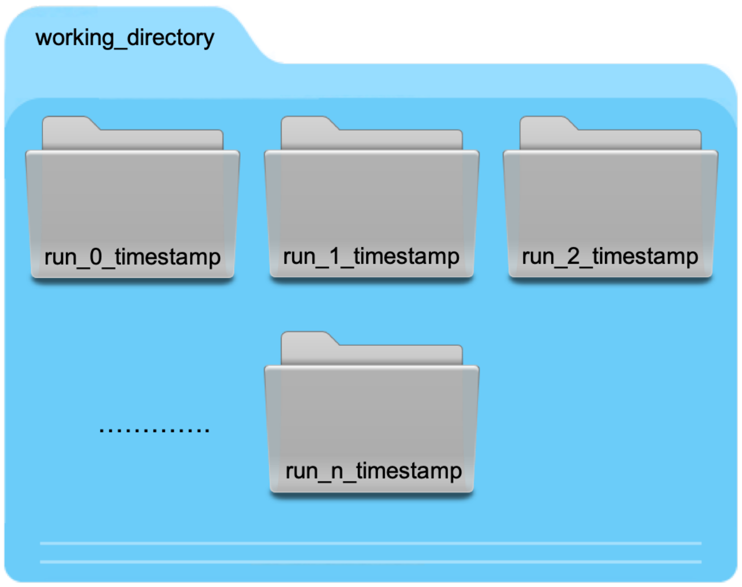


Figure 4: Within the working directory, `RunModel` creates folders, one for each sample input to the model. Each folder contains all the output corresponding to the model run with that input.

template through place-holder variables following a `UQpy` specific convention. This convention is described herein. The place-holder variables are replaced by `RunModel` with numerical values from the `samples` passed as input to `RunModel`.

- Place-holders are defined by using `< >` around the variable name within the template input file. The variable names are specified within `RunModel` using the `var_names` input. `RunModel` scans the text within the input template looking for place-holders with each variable name and places the values in the appropriate location in the model input file. For example, if the user passes `var_names = ['var1']` and `samples = [[5.2], [3.9], [4.4]]`, `RunModel` will generate three input files (one for each sample). In the first input file, the value of `5.2` replaces the place-holder `<var1>` where ever it appears in the the template input

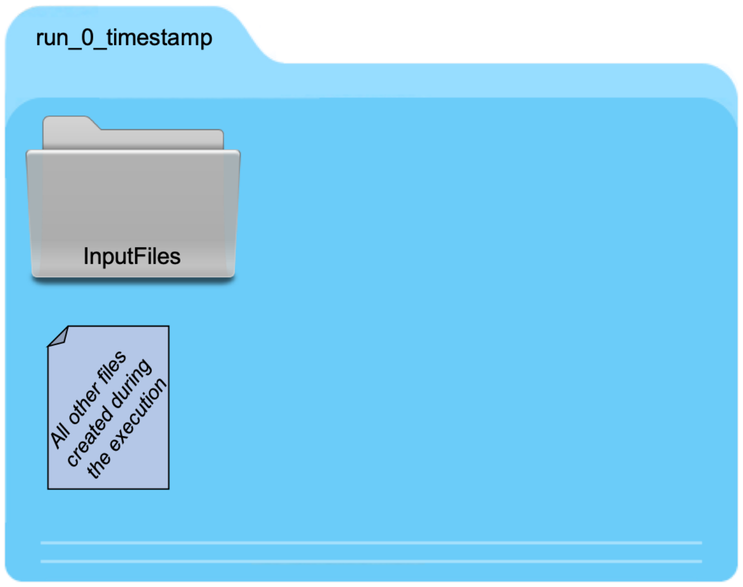


Figure 5: Within each directory corresponding to one sample, `RunModel` creates a folder called `InputFiles` which contains the input file generated using that sample, and all the outputs generated during the model execution using that sample.

file. In the second and third input files, `<var1>` is replaced by `3.9` and `4.4` respectively.

As previously stated, if `var_names = None`, `RunModel` assigns variable names as *x0, x1, x2, ..., xn*.

Standard python indexing is supported when using the place-holders i.e., if `var1` is an array, then it is possible to specify, for example, `<var1[0][2]>`, which will then use the corresponding component of `var1` at that location. If `var1` is an array and when no specific component of `var1` is specified within the place-holders, i.e., if in the input template, only `<var1>` is used, then the entire contents of `var1` are written in a comma-separated format at that location in the input file.

- When `RunModel` is executed, it generates one input file for each row /

item of `samples` using the template input file. The names of the input files are built by appending an underscore and the sample index between the filename and the extension of the template input file. These input files are moved to a subdirectory, named `InputFiles` of the current working directory.

- An example of the usage of a template input follows for a simple Matlab model. In this example, three input files are generated for three samples of a single variable.

  The template input file is given as:

  ```
  matlab_model.m
  x = <var1>;
  y = x^2;
  fid = fopen('y.txt','w');
  fprintf(fid, '%d', y);
  fclose(fid);
  ```

  `RunModel` is called as follows:

  ```
  x = RunModel(samples = [[1.1], [2.5], [3.3]], model_script
  = 'matlab_model_script.py', input_template = 'matlab_model.m',
  var_names = ['var1'], output_script = 'output.py',
  output_object_name = 'postprocess', ntasks = 1)
  ```

  When `RunModel` is executed, it then builds three input files as follows:

  ```
  matlab_model_1.m
  x = 1.1;
  y = x^2;
  fid = fopen('y.txt','w');
  fprintf(fid, '%d', y);
  fclose(fid);
  ```

```
matlab_model_2.m
x = 2.5;
y = x^2;
fid = fopen('y.txt','w');
fprintf(fid, '%d', y);
fclose(fid);
```

```
matlab_model_3.m
x = 3.3;
y = x^2;
fid = fopen('y.txt','w');
fprintf(fid, '%d', y);
fclose(fid);
```

These three files serve as input to the model that is evaluated by `model_script`, which is discussed next.

`model_script`:

`model_script` is the user-defined Python script that runs the computational model. It can be employed in two different ways depending on the type of model being executed.

- **Python Model**: The `model_script` should have defined within it an object (either a class object or a function object), specified in `RunModel` by `model_object_name`, which contains the computational model itself. In such a case, the `samples` passed to `RunModel` are passed as inputs to the model object. Refer to 5.1.3 for the structure of `model_script` in this case.

- **Third-party Software Model**: When running a third-party model, `RunModel` does not import `model_script`. Instead, `RunModel` calls the model script through the command line as

  `python3 model_script(sample_index)`

  using the Python `fire` module. Notice the only variable passed into `model_script` is the sample index. This is because the samples are being passed through the input files. For example, if the model object is passed the sample index n, it should then execute the model using the input file whose name is `input_n.inp`, where `input_template = input.inp`.

An example of the the `model_script` corresponding to execution of a Matlab model with `input_template = matlab_model.m`, as illustrated in the `input_template` example, is given below.

```
matlab_model_script.py
```

```python
import os
import fire

if __name__ == ‘__main__’:
    fire.Fire(model)

def model(sample_index):
    # Copy the input file into the cwd
    command1 = "cp ./InputFiles/matlab_model_"
             + str(index + 1) + ".m ."
    # Run the model
    command2 = "matlab -nosplash -nojvm -nodisplay
             -nodesktop -r ‘run matlab_model_"
             + str(sample_index + 1) + ".m; exit’"
    # Rename the output file
    command3 = "mv y.txt y_" + str(sample_index + 1)
             + ".txt"
    os.system(command1)
    os.system(command2)
    os.system(command3)
```

In `model_script` file, it is necessary to build the executable commands into a function (here called `model`) so that the sample index can be passed into the script – allowing the script to recognize which input file to use. Because the executable commands must be built into a function, it is necessary to call this function using the Python `fire` module as illustrated in the first two lines of `matlab_model_script.py`.

Again, `RunModel` is called as follows:

`x = RunModel(samples = [[1.1], [2.5], [3.3]], model_script = ‘matlab_model_script.py’, input_template = ‘matlab_model.m’,`

```
var_names = [‘var1’], output_script = ‘output.py’,
output_object_name = ‘postprocess’, ntasks = 1)
```

Also notice that the model script must index the name of the output file for subsequent postprocessing through the `output_script` as discussed next.

`output_script`:

- `output_script` is an optional user-defined Python script for post-processing model output. Specifically, it is used to extract user-specified quantities of interest from third-party model output files and return them to `RunModel`. `output_script` is used only when using `RunModel` with a third-party software model.

- `UQpy` imports the `output_script` and executes the object defined by `output_object_name`. The structure of the output object should be such that it accepts only the sample index as the input. If the model object is a Class, the quantity of interest must be stored as an attribute of the class `self.qoi`. If it is a function, it must return the quantity of interest after execution.

  In summary, if the output object is a class, it should be structured as follows:

  ```
  class OutputClass:
      def __init__(self, input=sample_index):
          Post-process the output files corresponding the the sample number
          and extract the quantity of interest.
          self.qoi = output
  ```

  or if it is a function, it should be structured as follows:

  ```
  def output_function(input=sample_index):
      Post-process the output files corresponding the the sample number
      and extract the quantity of interest.
      return output
  ```

  In keeping with the Matlab example illustrated for the `input_template` and `model_script`, an example `output_script` is given as follows:

```
output.py
def postprocess(sample_index):
     x = np.loadtxt("y_%d.txt" % (sample_index + 1))
     return x
```

Executable Software:
Often, the working directory will contain an executable software program. This is the case when the software does not lie in the user's path.

5.1.9 Examples & Template Files:

Examples illustrating the use of `RunModel` are provided in the following `Jupyter` notebooks.

- `Matlab_Model_Example.ipynb`:
  In this example, a small set of one dimensional random samples are drawn from a standard Normal distribution using the `MCS` class. Matlab is called to return the square of the random variable using the `RunModel` class.

- `Python_Model_Example.ipynb`:
  In this example, a set of 10,000 three-dimensional random samples are drawn from a standard Normal distribution using the `MCS` class. Two Python models, `python_model_class.py` and `python_model_function.py`, are called to sum each of the 10,000 random samples. The first model structures the Python model as a class and the second model structures the Python model as a function. Both models are run serially and in parallel.

A number of template scripts for commonly used third-party software applications are currently under development. These templates should not be considered as fully-functional software models (as is the case with the provided examples). Instead, they are meant to provide an initial starting point for users interested in linking `UQpy` with common software.

- Matlab
  Coming soon…

- Abaqus
  Coming soon…

- LS-DYNA
  Coming soon...
- OpenSEAS
  Coming soon...
- OpenFOAM
  Coming soon...
- FEAP
  Coming soon...
- SAFIR
  Coming soon...

5.2 `SampleMethods` Module

The `SampleMethods` module consists of classes to draw samples of random variables. It is imported in a python script using the following command:

```
from UQpy import SampleMethods
```

The `SampleMethods` module has the following classes, each corresponding to a different sampling method:

Class	**Method**
`MCS`	Monte Carlo Sampling
`LHS`	Latin Hypercube Sampling
`STS`	Stratified Sampling
`MCMC`	Markov Chain Monte Carlo
`IS`	Importance sampling
`RSS`	Refined Stratified Sampling
`Simplex`	Uniform Sampling on a Simplex

Each class can be imported individually into a python script. For example, the `MCMC` class can be imported to a script using the following command:

```
from UQpy.SampleMethods import MCMC
```

The following subsections describe each class, their respective inputs and attributes, and their use.

5.2.1 UQpy.SampleMethods.MCS

Theory
Monte Carlo sampling (MCS) generates independent random draws from a specified probability distribution or distributions. The MCS class utilizes the `scipy.stats` package for many predefined parametric distributions through the Distributions class (see Section 6.1). The user may also specify a custom distribution as outlined in Section 6.1.

The advantage of using the `MCS` class for `UQpy` operations, as opposed to simply generating samples with the `scipy.stats` package, is that it builds an object containing the samples, their distributions, parameters, and variable names for integration with other `UQpy` modules.

If `MCS` is used to generate multi-variate random vectors, the components of the vector will be independent and will therefore follow a product distribution. To induce correlation between components, use the `Transformations.Correlate` as described in Section 5.7.1.

Using the `MCS` Class
The `MCS` class is imported using the following command:

```
from UQpy.SampleMethods import MCS
```

The attributes of the `MCS` class are listed below:

MCS Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
`dist_name`	Input	⋆	
`dist_params`	Input	⋆	
`nsamples`	Input	⋆	
`var_names`	Input		⋆
`verbose`	Input		⋆
`samples`	Output		

A brief description of each attribute can be found in the table below:

MCS Class Attributes			
Attribute	**Type**	**Options**	**Default**
`dist_name`	*string* *string list*	See `Distributions` Module	`None`
`dist_params`	*ndarray* *list*	See `Distributions` Module	`None`
`nsamples`	*integer*		`None`
`var_names`	*string* *string list*		`None`
`verbose`	*boolean*		`False`
`samples`	*ndarray*		

Detailed Description of `MCS` Class Attributes:

Input Attributes:

- `dist_name`:
  Defines the name of the distribution for each random variable.

  `dist_name` may be a string or a list of strings.

  If `dist_name[i]` is a string, the distribution is matched with one of the available distributions in the `Distributions` module (see Sec. 6.1) or the user-defined custom distribution is called (again see Sec. 6.1).

  `dist_name` must be specified. There is no default value.

- `dist_params`:
  Specifies the parameters for each distribution in `dist_name`.

  Each set of parameters is defined as a numpy array. `dist_params` is a list of arrays, with each item in the list corresponding to the associated random variable.

  `dist_params` must be specified. There is no default value.

- `nsamples`:
  Specifies the number of samples to be generated as an integer.

  `nsamples` must be specified. There is no default value.

- `var_names`:
  Specifies the names of the random variables. Variable names are used as place-holders within input files for analyses driven by `RunModel`.

`var_names` is optional and should contain a list of strings of the same length as the number of random variables.

`var_names` has no default value.

- `verbose`:
  Specifies whether text is written to the terminal declaring the status of the `MCS` evaluation.

  `verbose` is of boolean type with default `verbose = False`.

Output Attributes:

- `samples`:
  A `numpy` array of dimension `nsamples` $\times$ `n`, where `n` is the number of random variables, containing the generated random samples following the specified distribution.

Examples:
Two examples illustrating the use of the `MCS` class are provided in the following Jupyter scripts.

- MCS_Example1.ipynb:
  In this example, 1000 2-dimensional samples are drawn from a standard normal distribution.

- MCS_Example2.ipynb:
  In this example, 1000 2-dimensional samples are drawn from a custom distribution (defined through custom_dist.py).

5.2.2 `UQpy.SampleMethods.LHS`

Theory
Latin hypercube sampling (LHS) belongs to the family of stratified sampling techniques and has the advantage that the samples generated are better distributed in the parameter space. LHS is perfomed by dividing the the range of each random variable into N bins with equal probability mass, where N is the required number of samples and then generating one sample per bin. Latin hypercube sampling has a faster convergence rate than crude Monte Carlo simulation and reduces the variance of statistical estimates.

Using the `LHS` Class
`LHS` is a class for Latin hypercube sampling. The `LHS` class is imported using the following command:

```
from UQpy.SampleMethods import LHS
```

The attributes of the LHS class are listed below:

LHS Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
`dimension`	Input		⋆
`dist_name`	Input	⋆	
`dist_params`	Input	⋆	
`lhs_criterion`	Input		⋆
`lhs_metric`	Input		⋆
`lhs_iter`	Input		⋆
`nsamples`	Input	⋆	
`samplesU01`	Output		
`samples`	Output		

A brief description of each attribute can be found in the table below:

LHS Class Attributes			
Attribute	**Type**	**Options**	**Default**
`dimensions`	*integer*		`dimension = len(dist_name)`
`dist_name`	*function/string list*	See `Distributions` Module or User-defined function	
`dist_params`	*ndarray list*		
`lhs_criterion`	*string*	'random' 'centered' 'maximin' 'correlate'	'random'
`lhs_metric`	*string*	'braycurtis', 'canberra', 'chebyshev' 'cityblock', 'correlation', 'cosine' 'dice','euclidean', 'hamming' 'jaccard', 'kulsinski', 'mahalanobis' 'matching', 'minkowski', 'rogerstanimoto' 'russellrao', 'seuclidean', 'sokalmichener' 'sokalsneath', 'sqeuclidean', 'yule'	'euclidean'
`lhs_iter`	*integer*		`iterations` = 100
`nsamples`	*integer*		`None`
`samplesU01`	*ndarray*		
`samples`	*ndarray*		

Detailed Description of LHS Class Attributes:

Input Attributes:

- `dimension`:
  A scalar integer value defining the dimension of the random variables.

- `dist_name`:
  Defines the distributions for each random variable.

  `dist_name` may be a string, a function, or a list of strings/functions.

  If `dist_name[i]` is a string, the distribution is matched with with one of the available functions in the `Distributions` module (see Sec. 6.1) or the 'custom_dist.py' (again see Sec. 6.1).

  if `dist_name[i]` is a function, it must be defined in the user's Python script and passed directly as a function.

  `dist_name` can contain an arbitrary combination of strings and functions.

  If `dist_name` is a string or function (or a list of length one) and `dimension` > 1, then `dist_name` is converted into a list of length `dimension` with each variable having the same distribution.

  `dist_name` must be specified. There is no default value.

- `dist_params`:
  Specifies the parameters for each distribution in `dist_name`.

  Each set of parameters is defined as a numpy array. `dist_params` is a list of arrays, with each item in the list corresponding to the associated random variable.

  If `dist_params` is an array (or a list of length one), then `dist_params` is converted to a list of length `dimension` with each variable having the same parameters.

  `dist_params` must be specified. There is no default value.

- `lhs_criterion`:
  Design criterion for the Latin hypercube samples. The different choices available are given below:

  - ‘random’: Samples are drawn randomly in the Latin hypercube strata.
  - ‘centered’: Samples are centered in the Latin hypercube strata.
  - ‘maximin’: The minimum distance between the sample points is maximized.
  - ‘correlate’: The correlation among the sample points is minimized.

- `lhs_metric`:
  Specifies the distance metric to be used in the case of ‘maximin’ criterion. The choices are the avaialable distance metrics in `scipy`.

  Only required in the case of `lhs_criterion` = ‘maximin’.

- `lhs_iter`:
  Specifies the number of iterations to be run for deciding the design in the case of `lhs_criterion` = ‘maximin’ and `lhs_criterion` = ‘correlate’.

- `nsamples`:
  Specifies the number of samples to be generated.

  `nsamples` must be specified. There is no default value.

Output Attributes:

- `samplesU01`:
  A numpy array of dimension $\texttt{nsamples} \times \texttt{dimension}$ containing the samples generated uniformly on the hypercube $[0,1]^{\texttt{dimension}}$.

- `samples`:
  A numpy array of dimension $\texttt{nsamples} \times \texttt{dimension}$ containing the samples following the specified distribution.

Examples:
An example illustrating the use of the `LHS` class is provided in the following Jupyter script.

- LHS.ipynb:
  In this example, 5 2-dimensional samples are drawn using Latin hypercube sampling with different `lhs_criterion` to illustrate its use.

5.2.3 `UQpy.SampleMethods.STS`

Theory

Stratified Sampling is a variance reduction sampling technique, it aims to distribute random samples on the complete sample space. Sample space is divided into exclusive groups, called strata and samples are generated inside each strata using uniform distribution.

Using the `STS` Class

`STS` is a class for stratified sampling. The `STS` class is imported using the following command:

```
from UQpy.SampleMethods import STS
```

The attributes of the `STS` class are listed below:

STS Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
`dimension`	Input		⋆
`dist_name`	Input	⋆	
`dist_params`	Input	⋆	
`sts_design`	Input		⋆
`sts_criterion`	Input		⋆
`input_file`	Input		⋆
`samples`	Output		
`samplesU01`	Output		
`strata`	Output		

A brief description of each attribute can be found in the table below:

`STS` Class Attributes			
Attribute	**Type**	**Options**	**Default**
`dimension`	*integer*		`dimension = len(sts_design)`
`dist_name`	*function/string list*	See `Distributions` Module or User-defined function	
`dist_params`	*ndarray list*		
`sts_design`	*int list*		`None`
`sts_criterion`	*string*	[‘random’, ‘centered’]	random
`input_file`	*string*		`None`
`samples`	*ndarray*		
`samplesU01`	*ndarray*		
`strata`	`class object`	See `UQpy.SampleMethods.Strata`	

Detailed Description of `STS` Class Attributes:

Input Attributes:

- `dimension`:
  A scalar integer value defining the dimension of the random variables. It is not required, if `sts_design` is defined.

- `dist_name`:
  Defines the distributions for each random variable.

  `dist_name` may be a string, a function, or a list of strings/functions.

  If `dist_name[i]` is a string, the distribution is matched with one of the available functions in the `Distributions` module (see Sec. 6.1) or the user defined function (again see Sec. 6.1).

  if `dist_name[i]` is a function, it must be defined in the user's Python script and passed directly as a function.

  `dist_name` can contain an arbitrary combination of strings and functions.

  If `dist_name` is a string or function (or a list of length one) and `dimension` > 1, then `dist_name` is converted into a list of length `dimension` with each variable having the same distribution.

  `dist_name` must be specified. There is no default value.

- `dist_params`:
  Specifies the parameters for each distribution in `dist_name`.

  Each set of parameters is defined as a numpy array. `dist_params` is a list of arrays, with each item in the list corresponding to the associated random variable.

  If `dist_params` is an array (or a list of length one), then `dist_params` is converted to a list of length `dimension` with each variable having the same parameters.

`dist_params` must be specified. There is no default value.

- `sts_design`:
  Specifies the number of strata in each dimension.

  `sts_design` specifies a stratification that breaks every dimension equally into a specified number of strata of the same size. For more complex strata geometries, the strata boundaries can be explicitly defined through a text input file. See `input_file` and the corresponding documentation in Section 5.2.4.

  `STS` places one sample in each stratum so the total number of samples drawn by `STS` is the product of the components of `sts_design`.

  Example: `sts_design = [2, 4, 3]` specifies a three-dimensional stratified design with two strata in the first dimension, four strata in the second dimension, and three strata in the third dimension for a total of $2 \times 4 \times 3 = 24$ samples.

- `sts_criterion`:
  It is a string specifying the technique used to generate sample inside each strata. A sample can be generated randomly or center of each stratum can be return as sample. 'random' generates sample using uniform distribution and 'centered' returns the center of each stratum. Default is 'random'.

- `input_file`:
  Specifies the file path of for a text file defining a stratification. See Section 5.2.4

Output Attributes:

- `samples`:
  The generated samples. The samples are returned as a numpy array.

- `samplesU01`:
  The untransformed samples drawn from the unit hypercube with dimension `dimension`.

- `strata`:
  A class object that defines the strata on the unit hypercube with dimension `dimension`.

Examples:
Two examples illustrating the use of the `STS` class are provided in the following Jupyter scripts.

- STS_Example1.ipynb:
  In this example, 25 samples are drawn from an exponential distribution using stratified sampling with the strata specified using the `sts_design` input for a regular, equal probability stratification.

- STS_Example2.ipynb:
  In this example, 6 samples are drawn from an exponential distribution using stratified sampling with the strata specified using an `input_file` (‘strata.txt) to create an irregular stratification with unequal probability strata.

5.2.4 `UQpy.SampleMethods.Strata`

The `Strata` class is a supporting class for stratified sampling and its variants. The class defines a rectilinear stratification of the unit hypercube. Strata are defined by specifying an origin as the coordinates of the stratum corner nearest to the origin and a stratum width for each dimension.
The attributes of the `STS` class are listed below:

`Strata` Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
`nstrata`	Input		⋆
`input_file`	Input		⋆
`origins`	Output		
`widths`	Output		
`weights`	Output		

A brief description of each attribute can be found in the table below:

`Strata` Class Attributes			
Attribute	**Type**	**Options**	**Default**
`nstrata`	*int list*		`None`
`input_file`	*string*		`None`
`origins`	*ndarray*		
`widths`	*ndarray*		
`weights`	*ndarray*		

Detailed Description of `Strata` Class Attributes:

Input Attributes:

- `nstrata`:
  Specifies the number of strata in each dimension. This is equivalent to `sts_design` from the `STS` class. For additional details, see `STS` documentation in Section 5.2.3.

  When calling the `Strata` class, the user must provide either `nstrata` or a text file defining the strata specified through `input_file`.

- `input_file`:
  Specifies the file path of for a text file defining a stratification.

  When calling the `Strata` class, the user must provide either `nstrata` or a text file defining the strata specified through `input_file`.

  File format: This file must be a space delimited text file having 2×`dimension` columns and the number of rows equal to the number of strata. The first `dimension` columns correspond to the coordinates in each dimension of the stratum origin. Columns `dimension+1` to 2×`dimension` correspond to the stratum widths in each dimension.

  For example, to specify stratification with two 2-dimensional strata, the text file might contain the following:

  ```
  0.0 0.0 0.5 1.0
  0.5 0.0 0.5 1.0
  ```

  The first stratum (row 1) has origin `(0.0, 0.0)` and has width `0.5` in dimension 1 and width `1.0` in dimension 2. The second stratum (row 2) has origin `(0.5, 0.0)` and has width `0.5` in dimension 1 and width `1.0` in dimension 2.

  When manually assigning the strata definitions, the user must be careful to ensure that the stratification fills the space without overlap. That is, each strata that the user defines must be disjoint and the total volume of the strata must be equal to one (i.e. it must fill the unit hypercube).

An example `input_file` can be found in 'STS_Example2' in the provided example Jupyter scripts.

Output Attributes:

- `origins`:
  Specifies the coordinates of the origin of each stratum.

- `widths`:
  Specifies the width in each dimension of each stratum.

- `weights`:
  The volume of each stratum (`=prod(widths)` for each stratum), `weights` are the probabilities assigned to each sample in a stratified sample design.

5.2.5 `UQpy.SampleMethods.MCMC`

Theory

The goal of Markov Chain Monte Carlo is to draw samples from some probability distribution $p(x) = \frac{\tilde{p}(x)}{Z}$, where $\tilde{p}(x)$ is known but Z is hard to compute (this will often be the case when using Bayes' theorem for instance). In order to do this, the theory of a Markov chain, a stochastic model that describes a sequence of states in which the probability of a state depends only on the previous state, is combined with a Monte Carlo simulation method. More specifically, a Markov Chain is built and sampled from whose stationary distribution is the target distribution $p(x)$. The reader is referred to e.g. [6] for more theory about MCMC methods. The Metropolis-Hastings (MH) algorithm goes as follows:

- initialize with a seed sample x_0
- walk the chain: for $k = 0, ...$ do:
  - sample candidate $x^\star \sim Q(\cdot|x_k)$ for a given Markov transition probability Q
  - accept candidate (set $x_{k+1} = x^\star$) with probability

$$\alpha(x^\star|x_k) := min\{\frac{\tilde{p}(x^\star)}{\tilde{p}(x)} \cdot \frac{Q(x|x^\star)}{Q(x^\star|x)}, 1\}$$

    otherwise propagate last sample $x_{k+1} = x_k$

UQpy supports MH along with more advanced algorithms such as Modified Metropolis Hastings (MMH, [2]) and the Affine invariant ensemble sampler ([7]). The transition probability Q is chosen by the user (see inputs `pdf_proposal_type` and `pdf_proposal_scale`), and careful attention must be given to that choice as it plays a major role in the accuracy and efficiency of the algorithm. Figure 6 shows samples accepted (blue) and rejected (red) when trying to sample from a 2d Gaussian distribution using MH, for different scale parameters of the proposal distribution. If the scale is too small, the space is not well explored; if the scale is too large, many candidate samples will be rejected, yielding a very inefficient algorithm. As a rule of thumb, an acceptance ratio of 10%-50% could be targeted (see `Diagnostics` in the `Utilities` module).

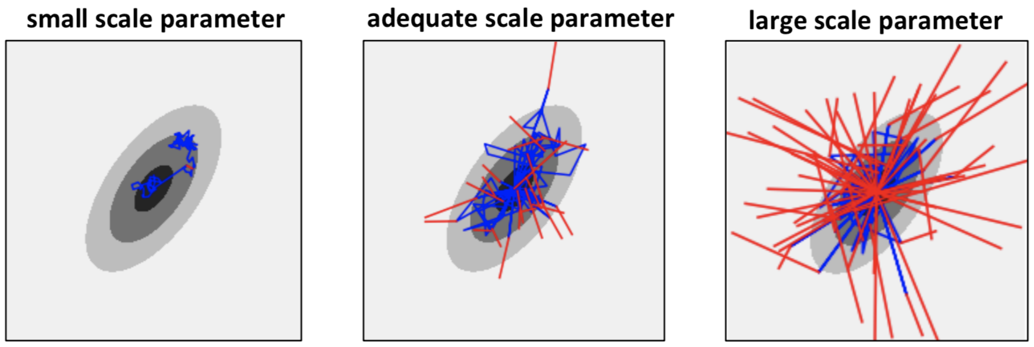


Figure 6: Sampling from a 2d Gaussian pdf using the MH algorithm and various scale parameters of the transition probability Q: in blue are the accepted draws from the Markov chain, in red the draws that were rejected.

Finally, samples from the target distribution will be generated only when the chain has converged to its stationary distribution, after a so-called burn-in period. Thus the user would often reject the first few samples (see input `burn`). Also, the chain yields correlated samples; thus to obtain i.i.d. samples from the target distribution, the user should keep only one out of `jump` samples (see input `jump`). This means that the code will perform in total `burn+jump*nsamples` evaluations of the target pdf to yield `nsamples` i.i.d. samples from the target distribution (for the MH algorithm).

In UQpy, the `MCMC` class is imported using the following command:

```
from UQpy.SampleMethods import MCMC
```

The attributes of the `MCMC` class are listed below:

`MCMC` Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
`dimension`	Input	⋆	
`pdf_proposal_type`	Input		⋆
`pdf_proposal_scale`	Input		⋆
`pdf_target`[1]	Input	⋆	
`log_pdf_target`[1]	Input	⋆	
`pdf_target_params`	Input		⋆
`pdf_target_copula`	Input		⋆
`pdf_target_copula_params`	Input		⋆
`pdf_target_type`	Input		⋆
`algorithm`	Input		⋆
`jump`	Input		⋆
`nsamples`	Input	⋆	
`seed`	Input		⋆
`nburn`	Input		⋆
`samples`	Output		
`accept_ratio`	Output		

A brief description of each attribute can be found in the table below:

*One of `pdf_target` or `log_pdf_target` is required.

MCMC Class Attributes			
Attribute	**Type**	**Options**	**Default**
`dimension`	*integer*		`dimension` = 1
`algorithm`	*string*	'MH' 'MMH' 'Stretch'	'MH'
`pdf_proposal_type`	*string*	'Normal' 'Uniform'	'Normal'

`pdf_proposal_scale`	*float*
	float list
			if `algorithm` = 'MMH' or 'MH':
			`pdf_proposal_scale` =[1,1,...,1]
			if `algorithm`='Stretch':
			`pdf_proposal_scale` = 2

`pdf_target`	*function* *string*		
`log_pdf_target`	*function*		`None`
`pdf_target_params`	*float* *float list*		`None`
`pdf_target_copula`	*str*		`None`
`pdf_target_copula_params`	*float* *float list*		`None`
`pdf_target_type`	*string*	'marginal_pdf' 'joint_pdf'	only used if `algorithm` = 'MMH'
`jump`	*integer*		1
`nsamples`	*integer*		`None`

`seed`	*ndarray*
	ndarray list
			array(0,0,...,0)
			size = $1 \times$ `dimension`

`nburn`	*integer*		0
`samples`	*ndarray*		
`accept_ratio`	*float*		

Detailed Description of `MCMC` Class Attributes:

Input Attributes:

- `dimension`:
  A scalar integer value defining the dimension of the random variables.

- `algorithm`:
  Specifies the algorithm used to generate samples. `UQpy` currently supports three commonly used algorithms.

  - 'MH':
    Metropolis-Hastings algorithm. For a description of the algorithm, see [10, 9, 2].

  - 'MMH':
    Component-wise modified Metropolis-Hastings algorithm. For a description of the algorithm, see [2].

  - 'Stretch':
    Affine invariant ensemble sampler employing "stretch" moves. For a description of the algorithm, see [7].

- `pdf_proposal_type`:
  Type of proposal density function. This option is only invoked when `algorithm` = 'MH' or 'MMH'. `UQpy` currently supports two types of proposal densities:

  - 'Normal' (default):
    The proposal density is specified as a normal distribution with mean value equal to the current state of the Markov Chain and standard deviation specified by `pdf_proposal_scale`. That is, a new candidate sample is generated as
    $x_{i+1} \sim N(x_i, \texttt{pdf\_proposal\_scale})$.
  - 'Uniform':
    The proposal density is specified as a uniform distribution with centered at the current state of the Markov Chain with width equal to `pdf_proposal_scale`. That is, a new candidate sample is generated as
    $x_{i+1} \sim U(x_i - \texttt{pdf\_proposal\_scale}/2, x_i + \texttt{pdf\_proposal\_scale}/2)$.

  When `dimension` > 1, `pdf_proposal_type` may be specified as a string or a list of strings assigned to each dimension. When `pdf_proposal_type` is specified as a string, the same proposal type is specified for all dimensions.

- `pdf_proposal_scale`:
  Sets the scale of the proposal probability density. The scale of the proposal density depends on both the MCMC algorithm employed (`algorithm`) and the type of proposal density specified (`pdf_proposal_type`).

  - For `algorithm` = 'MH' or 'MMH', this defines either the standard deviation of a normal proposal density or the width of a uniform density. See `pdf_proposal_type` above.
  - For `algorithm` = 'Stretch', this sets the scale of the stretch density $g(z) = \frac{1}{\sqrt{z}}, \sim z \in [1/\texttt{pdf\_proposal\_scale}, \texttt{pdf\_proposal\_scale}]$. See [7].

When `dimension` > 1, `pdf_proposal_scale` may be specified as a scalar or a list of values assigned to each dimension. When `pdf_proposal_scale` is specified as a scalar, the same scale is specified for all dimensions.

- `pdf_target_type`:

  [Use only with `algorithm` = 'MMH']

  MCMC algorithms use acceptance-rejection based on a ratio of the target probability densities between the current state and the proposed state. In the 'MH' algorithm and the 'Stretch' algorithm, the ratio of probabilities is computed using the target joint pdf. For the 'MMH' algorithm with independent random variables, acceptance/rejection can be computed based on the ratio of the marginals for each dimension. This variable specifies whether to use a ratio of target joint pdf's or a ratio of target marginal pdf's in the acceptance-rejection step for each dimension of the 'MMH' algorithm. This option is not used for the 'MH' and 'Stretch' algorithms.

  - 'joint_pdf':
    Compute the acceptance-rejection using the ratio of the target joint pdf's. [Always use when random variables are dependent.]
  - 'marginal_pdf':
    Compute the acceptance-rejection using the ratio of target marginal pdf's in each dimension. [Only use when random variables are independent.]

- `log_pdf_target`:
  Specifies the density function p (or equivalently $\tilde{p}$), from which to draw MCMC samples `log_pdf_target` can be either:

  - a function (or list of functions for marginals):
    The easiest way to define `log_pdf_target` is to pass it as a function, or log$_p$*dfmethodofa*`Distribution`*classinstance.Thisfunctionmusttakeasinputparame*

    Inthiscase,a`Distribution`*instancewillbecreatedusingp* = `Distribution`(*dist_name* = `log_pdf_target`)*,andits*`log_pdf`*methodwillbecalledtoevalu*
    `The distribution can also accept a copula. If the built distribution p does not have a log_pdf method, an error is raised.`

Alternatively to specifying `log_pdf_target`, the user can specify `pdf_target`, see following item. However, for stability reasons (pdf values can become very small for unlikely draws), the algorithm always uses log pdfs instead of pdfs, thus, if possible, providing a log pdf function instead of a pdf is preferred. Figure 7 shows how the code checks the existence of a log_pdf or pdf callable that is used to evaluate $\log(\tilde{p}(x))$.

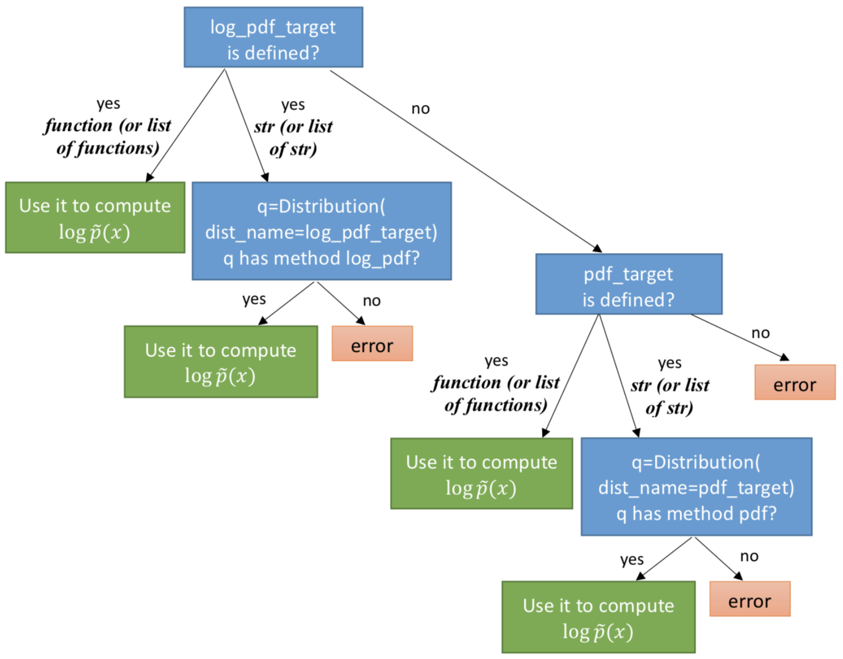

log_pdf_target is defined?
yes *function (or list of functions)*
yes *str (or list of str)*
no
Use it to compute $\log\tilde{p}(x)$
q=Distribution(dist_name=log_pdf_target) q has method log_pdf?
yes
no
Use it to compute $\log\tilde{p}(x)$
error
pdf_target is defined?
yes *function (or list of functions)*
yes *str (or list of str)*
no
error
Use it to compute $\log\tilde{p}(x)$
q=Distribution(dist_name=pdf_target) q has method pdf?
yes
no
Use it to compute $\log\tilde{p}(x)$
error

Figure 7: Diagram explaining how the code checks for the existence of the target distribution, used to evaluate $\log(\tilde{p}(x))$.

- `pdf_target`:
  Specifies the target probability density function from which to draw MCMC samples, alternative to defining `log_pdf_target`. `pdf_target` can be either:

  – a function (or list of functions for marginals):
    The easiest way to define `pdf_target` is to pass it as a function, or `pdf` method of a `Distribution` class instance. This function must take as input parameter at least one input `x`, the point where to evaluate the pdf, and can additionally take as input parameters `params, copula_params`.

  - a string (or list of strings for marginals):
    In this case, a `Distribution` instance will be created using p=`Distribution`(dist_name=`pdf_target`), and its `pdf` method will be called to evaluate $\log(\tilde{p}(x))$. The distribution can also accept a copula. If the built distribution p does not have a `log_pdf` method, an error is raised.

  When `dimension` $>$ 1 and `pdf_target_type` = 'marginal_pdf', `pdf_target` may be specified as a string/function or a list of strings/functions assigned to each dimension. When specified as a string/function, the same marginal pdf is specified for all dimensions.

- `pdf_target_params`:
  Parameters of the target pdf to be passed as arguments to the function defined by `pdf_target`, `log_pdf_target`.

- `pdf_target_copula`:
  Copula name of the target pdf if it exists. Used only if `pdf_target`, `log_pdf_target` are defined using strings/list of strings.

- `pdf_target_copula_params`:
  Parameters of the copula of the target pdf to be passed as arguments to the function defined by `pdf_target`, `log_pdf_target`.

- `jump`
  Specifies the number of samples between accepted states of the Markov chain. Setting `jump` $= 1$ corresponds to accepting every state. Setting `jump` $= n$ corresponds to skipping $n - 1$ states between accepted states of the chain.

- `nburn`
  Specifies the number of samples at the start of the chain to be discarded as "burn-in." This option is only applicable for `algorithm`='MMH' and 'MH'.

- `nsamples`
  Specifies the number of samples to be generated (not including the discarded burn-in states nor the skipped states of the chain). `nsamples` must be specified. There is no default value.

- `seed`
  Specifies the initial state of the Markov chain.

For `algorithm` = ‘MMH’ or ‘MH’, this is a numpy array of size $1 \times$ `dimension`. The default is a $1 \times$ `dimension` array of zeros.

For `algorithm` = ‘Stretch’, this is a list of n_s points, each defined as numpy arrays with size $1 \times$ `dimension`, where n_s is the size of the ensemble being propagated. [7]. The default value in the table above is not valid for `algorithm` = ‘Stretch’.

Output Attributes:

- `samples`:
  The generated samples are returned as a numpy array of dimension `nsamples` $\times$ `dimension`.

- `accept_ratio`:
  Acceptance ratio of the chain, an acceptance ratio between 10 and 50% could be targeted, see `Diagnostics`.

Examples:
Two examples illustrating the use of the `MCMC` class are provided in the following Jupyter scripts.

- MCMC_Example1.ipynb:
  In this example, the three MCMC algorithms are used to generate 1000 samples from a two-dimensional Rosenbrock pdf. The Rosenbrock pdf is defined as a function directly in the script, using both the `pdf_target` and `log_pdf_target` input parameters of the `MCMC` class.

- MCMC_Example2.ipynb:
  In this example, the three MCMC algorithms are used to generate 1000 samples from a two-dimensional Rosenbrock pdf. The Rosenbrock pdf is passed into the `MCMC` class as a string.

5.2.6 `UQpy.SampleMethods.IS`

Theory
Importance sampling (IS) is based on the idea of concentrating the distribution of the sampling points in regions of the input space. This allows to compute expectations $E_{\mathbf{x}\sim p}\left[f(\mathbf{x})\right]$ where $f(\mathbf{x})$ is small outside of a small region of the input space; thus the need to focus sampling around that small region. To this end, a sample $\mathbf{x}$ is drawn from a proposal distribution $q(\mathbf{x})$ and re-weighted to correct for the discrepancy between the sampling

distribution q and the true distribution p. The weight of the sample $\mathbf{x}$ is estimated as $\mathbf{w}(\mathbf{x}) = p(\mathbf{x})/q(\mathbf{x})$, where the quantity $p(\cdot)/q(\cdot)$ is called the likelihood ratio. In the case where p is only known up to a constant, i.e., one can only evaluate $\tilde{p}(\mathbf{x})$, where $p(\mathbf{x}) = \frac{\tilde{p}(\mathbf{x})}{Z}$, IS can be used by further normalizing the weights (self-normalized IS). Figure 8 shows the weighted samples obtained when using IS to estimate a 2d Gaussian target distribution p, sampling from a uniform proposal distribution q.

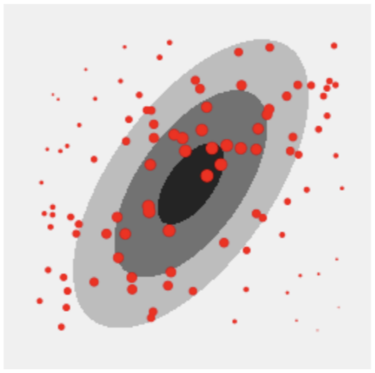


Figure 8: IS: samples are generated from a uniform distribution, then weighted to provide an approximation of the target Gaussian distribution.

Using the `IS` Class

The `IS` class is imported using the following command:

```
from UQpy.SampleMethods import IS
```

The attributes of the `IS` class are listed below:

`IS` Class Attribute Definitions				
Attribute	**Input/Output**	**Required**	**Optional**	**Type**
`nsamples`	Input	⋆		*integer*
`pdf_proposal`	Input	⋆		*string, strings list*
`pdf_proposal_params`	Input		⋆	*list* *list/ndarray list*
`log_pdf_target`†	Input	⋆		*string, strings list* *function, functions list*
`pdf_target`†	Input	⋆		*string, strings list* *function, functions list*
`pdf_target_params`	Input		⋆	*list* *list/ndarray list*
`pdf_target_copula`	Input		⋆	*str*
`pdf_target_copula_params`	Input		⋆	*list str*
`samples`	Output			*ndarray*
`weights`	Output			*ndarray*
`unnormalized_log_weights`	Output			*ndarray*

Detailed Description of IS Class Attributes:

Input Attributes:

- `pdf_proposal`:
  A string or list of strings providing the names of the proposal distribution (or its independent marginals) from which to sample. The distribution is then built as p=`Distribution`(dist_name=`pdf_proposal`). This distribution must have an `rvs` method, as well as a `log_pdf` (or `pdf`) method.

- `pdf_proposal_params`:
  Parameters of the proposal pdf, used when calling the `rvs` and `log_pdf` methods of the proposal distribution.

- `log_pdf_target`: This input defines the log of the target pdf $\log(\tilde{p}(x))$, it can either be:

  - a string or list of strings providing the names of the proposal distribution (or its independent marginals), then `Distribution` will be called. This `Distribution` instance must have a `log_pdf` method.
  - a function that evaluates the target pdf, given a matrix of samples x. This function must take in as input parameters at least one input `x`, namely the samples where to evaluate the log pdf; the function must be able to evaluate the log pdf of several samples at once, i.e., for an input `x` of size (nsamples, dimension), the function must return nsamples values of the log pdf. Additionally, it can take as inputs the parameters of the density functions `params` and copula parameters `copula_params`.

  Alternatively, the target pdf can be defined using `pdf_target`, the reader is referred to Figure 7 from the `MCMC` class for more detailed explanations on how the code checks for the definition of the target distribution.

- `pdf_target`: Alternative to defining `log_pdf_target`. This input can either be:

  - a string or list of strings providing the names of the proposal distribution (or its independent marginals), then `Distribution` will be called. This `Distribution` instance must have a `log_pdf` or a `pdf` method.

[†] One of `pdf_target` or `log_pdf_target` is required.

  - a function that evaluates the target pdf, given a matrix of samples x. Same comments apply as for `log_pdf_target` in this case.

- `pdf_target_params`:
  Parameters of the proposal pdf to be passed as arguments the target distribution.

- `pdf_copula`:
  Name of the copula of the target pdf, if it exists, used only if the input `pdf_target` is defined as a list of strings.

- `pdf_target_copula_params`:
  Parameters of the copula of the target pdf, if it exists, to be passed as arguments the target distribution.

- `nsamples`
  Specifies the number of samples to be generated. `nsamples` must be specified, there is no default value.

Output Attributes:

- `samples`:
  The samples of the `IS` class are returned as a numpy array of dimension `nsamples` $\times$ `dimension`.

- `weights`:
  The weights of the `IS` class are returned as a numpy array of dimension `nsamples`.

- `unnormalized_weights`:
  The logarithm of the unnormalized weights of the `IS` class are returned as a numpy array of dimension `nsamples`.

Examples:
One example illustrating the use of the `IS` class are provided in the following Jupyter script.

- IS_Example1.ipynb:
  In this example, IS is used to generate 500000 samples from a two-dimensional Rosenbrock pdf from a Uniform proposal distribution. The Rosenbrock pdf is defined as a function directly in the script.

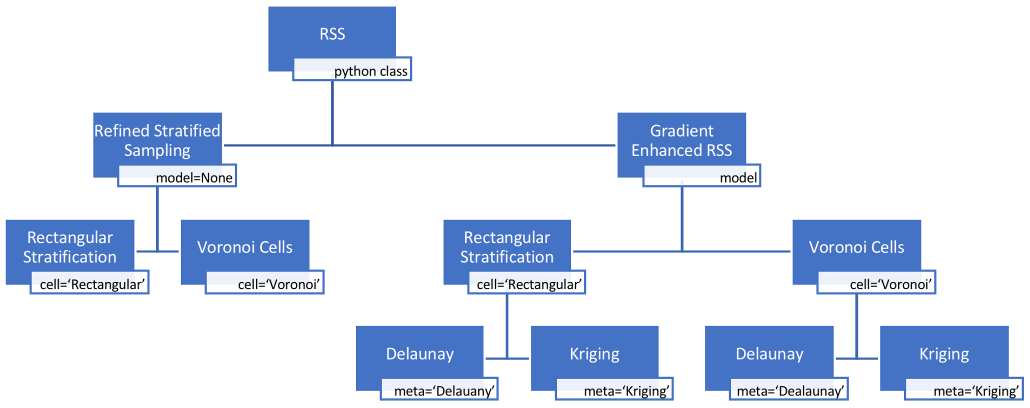


Figure 9: Work flow of RSS class.

5.2.7 `UQpy.SampleMethods.RSS`

Theory

This is a sample extension method, which uses random or gradient-based adaptive approach to reduce the variance of output random variable. This class divides sample domain using either rectangular stratification or voronoi cells. Fig(9) shows the work-flow of RSS class for different inputs attributes.

- Refined Stratified Sampling
  Randomly selects from the strata/cells with maximum weight, see paper [13] for detailed explanation.

- Gradient-Enhaced Refined Stratified Sampling
  Selects the strata/cells with maximum stratum variance, which is computed using Eq.(1), see [12] for detailed explanation.

$$\hat{\sigma}_j^2 \approx \nabla f(x_j^*)^T . \Sigma . \nabla f(x_j^*) . V_j \qquad \forall\ j \tag{1}$$

In case of rectangular stratification, selected strata is divided along the maximum width to define new strata. In case of voronoi cells, selected simplex is reduced down to sub-simplex, which is used for refinement.

Using the `RSS` Class

The `RSS` class is imported using the following command:

```
from UQpy.SampleMethods import RSS
```

The attributes of the RSS class are listed below:

RSS Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
x	Input	⋆	
model	Input		⋆
meta	Input		⋆
cell	Input		⋆
nsamples	Input	⋆	
min_train_size	Input		⋆
step_size	Input		⋆
corr_model	Input		⋆
corr_model_params	Input		⋆
reg_model	Input		⋆
n_opt	Input		⋆
samples	Output		
values	Output		

A brief description of each attribute can be found in the table below:

RSS Class Attributes			
Attribute	**Type**	**Options**	**Default**
x	*class*		None
model	*python script*		None
meta	*string*	Delaunay Kriging	Delaunay
cell	*string*	Rectangular Voronoi	Rectangular
nsamples	*int*		None
min_train_size	*int*		nsamples
step_size	*float*		0.005
corr_model	*string* *function*		Gaussian
corr_model_params	*ndarray*		[1, 1,..., 1]
reg_model	*string*		Quadratic
n_opt	*int*		1
samples	*ndarray*		
values	*ndarray*		

Detailed Description of `RSS` Class Attributes:

Input Attributes:

- `x`:
  A class object generated using `STS` or `RSS` class. It contains the information about coordinates, stratification and weights corresponding to existing samples. This class requires an initial `STS` design to function.

- `model`
  A string specifying the python script, which is used to evaluate model at sample points. It is called with RunModel, see section 5.1.3 for detailed explanation. It is required for GE-RSS, if `model` is 'None' Refined Stratified Sampling is executed for sample expansion.

- `meta`
  A string specifying the method used to estimate gradient of function. '`Delaunay`' creates a linear interpolator over the domain, whereas, `Kriging`' generates an approximate surrogate model. It is only required for GE-RSS method. Default string is `Delaunay`.

- `cell`
  A string specifying the stratification of sample space. This class supports two types of stratification, i.e. Rectangular and Voronoi. Default string is `Rectangular`.

- `nsamples`
  An integer specifying the final size of extended samples.

- `min_train_size`
  An integer specifying the minimum number of samples used to generate local surrogate model to update gradient of the function. Only required if kriging surrogate is used to estimate gradient.

- `step_size`
  A real number defining the step size to calculate the gradient using central difference method.

- `corr_model`
  A string specifying the correlation model used to create the surrogate model. Only required if kriging surrogate is used to estimate gradient, see section 5.5.2 for details.

- `corr_model_params`
  An array specifying initial values corresponding to hyperparameters/scale parameters. Only required if kriging surrogate is used to estimate gradient, see section 5.5.2 for details.

- `reg_model`
  A string specifying the regression model used to create the surrogate model. Only required if kriging surrogate is used to estimate gradient, see section 5.5.2 for details.

- `n_opt`
  Number of times optimization problem is to be solved with different starting point, see section 5.5.2 for details. Here, this is done for only first sample, after that hyperparameter from previous kriging is used as starting point. Default: 1

Output Attributes:

- `samples`:
  The samples of the `RSS` class are returned as a numpy array of dimension `nsamples` × `dimension`. Dimension is same as of samples in object x.

- `values`:
  The values of the `RSS` class are returned as a numpy array. It is the function value at the sample points evaluated using RunModel.

Examples:
One example illustrating the use of the `RSS` class are provided in the following Jupyter script.

- RSS_Example1.ipynb:
  This example demonstrate the use of Refined Stratified Sampling with rectilinear stratification through `RSS` class. First, The `STS` is used to generate 16 samples using uniform probability distribution. `RSS` class is used to extend samples to 18 points. Plots illustrates the modified stratification with new samples. Further, samples from `RSS` class have been used again to expand samples to 100 points.

- RSS_Example2.ipynb:
  This example demonstrate the use of Refined Stratified Sampling with voronoi stratification. First, The `STS` is used to generate 16 samples using uniform probability distribution. `RSS` class is used to extend samples to 18 points. Plots illustrates the modified stratification with new samples.

Further, samples from `RSS` class have been used again to expand samples to 100 points.

- RSS_Example3.ipynb:
  This example illustrate the use of Gradient Enhanced Refined Stratified Sampling with rectilinear stratification. 'LinearNDInterpolator' is used to estimate the gradient. `RSS` class expands the 16 samples from `STS` class to 200 samples.

- RSS_Example4.ipynb:
  This example illustrate the use of Gradient Enhanced Refined Stratified Sampling with rectilinear stratification. '`Krig`' class is used to estimate the gradient. `RSS` class expands the 16 samples from `STS` class to 200 samples.

- RSS_Example5.ipynb:
  This example illustrate the use of Gradient Enhanced Refined Stratified Sampling with voronoi stratification. '`Krig`' class is used to estimate the gradient. `RSS` class expands the 16 samples from `STS` class to 100 samples.

- RSS_Example6.ipynb:
  This example illustrate the use of Gradient Enhanced Refined Stratified Sampling with voronoi stratification. '`Krig`' class is used to estimate the gradient. `RSS` class expands the 16 samples from `STS` class to 100 samples.

5.2.8 `UQpy.SampleMethods.Simplex`

Theory

Edeling et al. [5] discuss the method to generate uniformly distributed sample inside a simplex, whose coordinates are expressed by ζ_k and n_d is dimension. First, generate n_d independent uniform random variables on [0, 1], i.e. r_q, then compute

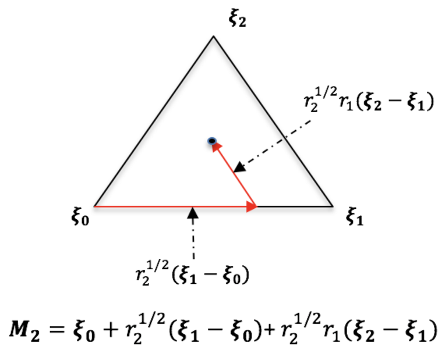
$$\mathbf{M_{n_d}} = \boldsymbol{\zeta_0} + \sum_{i=1}^{n_d} \Bigg[\prod_{j=1}^{i} r_{n_d-j+1}^{\frac{1}{n_d-j+1}}\Bigg](\boldsymbol{\zeta_i} - \boldsymbol{\zeta_{i-1}})$$

The M_{n_d} is n_d dimensional array defining the coordinates of new sample.

Using the `Simplex` Class

The `Simplex` class is imported using the following command:

```
from UQpy.SampleMethods import Simplex
```



ξ_2
$r_2^{1/2} r_1(\xi_2 - \xi_1)$
ξ_0
ξ_1
$r_2^{1/2}(\xi_1 - \xi_0)$
$M_2 = \xi_0 + r_2^{1/2}(\xi_1 - \xi_0) + r_2^{1/2} r_1(\xi_2 - \xi_1)$

Figure 10: Random point inside a 2-D Simplex.

The attributes of the `Simplex` class are listed below:

`Simplex` Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
`nodes`	Input	⋆	
`nsamples`	Input	⋆	
`samples`	Output		

A brief description of each attribute can be found in the table below:

`Simplex` Class Attributes			
Attribute	**Type**	**Options**	**Default**
`nodes`	*ndarray/list*		`None`
`nsamples`	*integer*		`1`
`samples`	*ndarray*		

Detailed Description of `Simplex` Class Attributes:

Input Attributes:

- `nodes`:
  An array or list defining the coordinates of the vertices of simplex. This is a required attribute, there is no default value.

- `nsamples`
  Specifies the number of samples to be generated. `nsamples` must be specified. Default value is 1.

Output Attributes:

- `samples`:
  The samples of the `Simplex` class are returned as a numpy array of dimension `nsamples` $\times$ `dimension`. Dimension is equal to number of vertex - 1.

Examples:
One example illustrating the use of the `Simplex` class is provided in the following Jupyter script.

- Simplex_Example1.ipynb:
  In this example, Simplex class is used to generate 10 samples inside two-dimensional simplex from a Uniform proposal distribution.

5.3 `Inference` Module

The goal in inference can be twofold: 1) given a model, parameterized by parameter vector θ, and some data $\mathcal{D}$, learn the value of the parameter vector that best explains the data; 2) given a set of candidate models $\{m_i\}_{i=1:M}$ and some data $\mathcal{D}$, learn which model best explains the data. UQpy supports the following inference algorithms for parameter estimation:

- MLEstimation (parameter estimation by maximum likelihood, frequentist approach),
- BayesParamEstimation (parameter estimation using MCMC or IS, Bayesian approach).

and the following algorithms for model selection:

- InfoModelSelection (model selection using information theoretic criteria),
- BayesModelSelection (Bayesian model class selection).

The capabilities of `UQpy` and associated classes are summarized in Fig. 11.

5.3.1 `UQpy.Inference.Model`

In all cases, the user must first create, for each model studied, an instance of the class Model, which can be either:

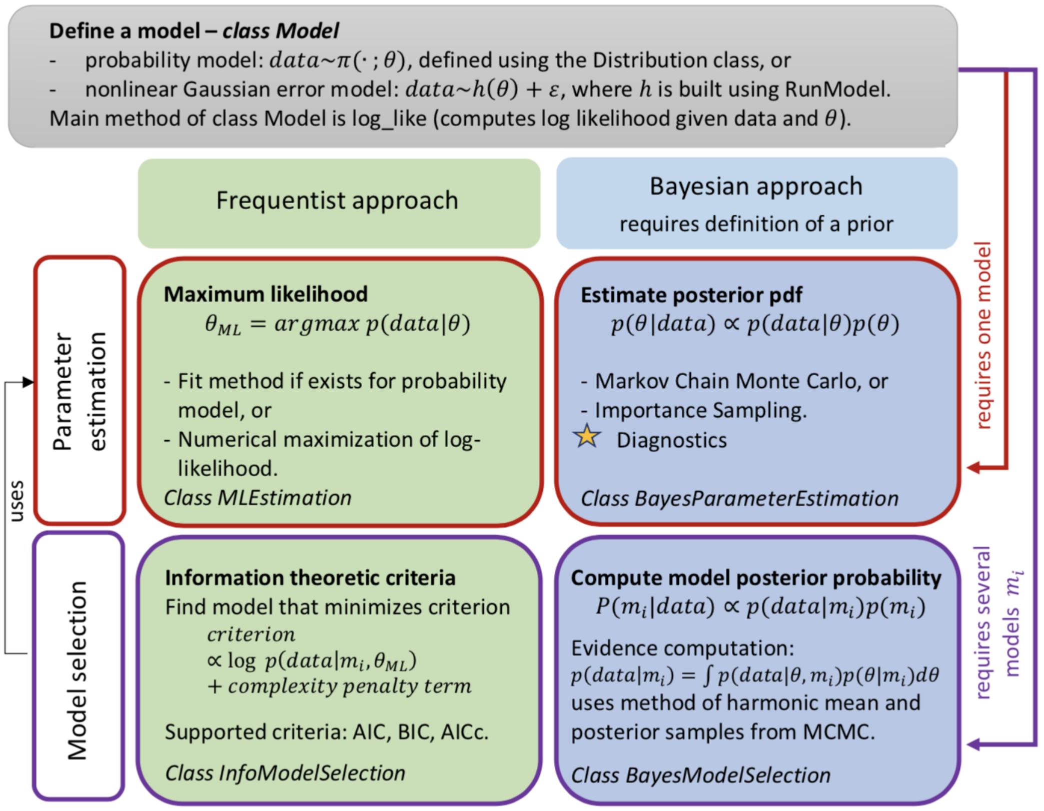

Define a model – class Model
- probability model: $data \sim \pi(\cdot\,; \theta)$, defined using the Distribution class, or
- nonlinear Gaussian error model: $data \sim h(\theta) + \varepsilon$, where h is built using RunModel.
Main method of class Model is log_like (computes log likelihood given data and θ).
Frequentist approach
Bayesian approach
requires definition of a prior
Parameter estimation
Model selection
uses
Maximum likelihood
$\theta_{ML} = argmax\ p(data|\theta)$
- Fit method if exists for probability model, or
- Numerical maximization of log-likelihood.
Class MLEstimation
Estimate posterior pdf
$p(\theta|data) \propto p(data|\theta)p(\theta)$
- Markov Chain Monte Carlo, or
- Importance Sampling.
Diagnostics
Class BayesParameterEstimation
requires one model
Information theoretic criteria
Find model that minimizes criterion
$criterion \propto \log\ p(data|m_i, \theta_{ML}) + complexity\ penalty\ term$
Supported criteria: AIC, BIC, AICc.
Class InfoModelSelection
Compute model posterior probability
$P(m_i|data) \propto p(data|m_i)p(m_i)$
Evidence computation:
$p(data|m_i) = \int p(data|\theta, m_i)p(\theta|m_i)d\theta$
uses method of harmonic mean and posterior samples from MCMC.
Class BayesModelSelection
requires several models m_i

Figure 11: UQpy Inference module.

- a probability model π, where $\mathcal{D} \sim \pi(\cdot|\theta)$; π is a distribution defined using the Distribution module;
- a user-defined model $h(\theta)$ given in a python script (see requirements in the RunModel section). The associated probabilistic model for inference is defined as $\mathcal{D} = h(\theta) + \epsilon$, where the error ϵ is assumed to be Gaussian with zero mean.

The class defines a `log_like` method as a function that evaluates, given a data vector $\mathcal{D}$ and a parameter vector θ, the log likelihood of the data $\ln p(\mathcal{D}|\theta)$. For a probability model, $\mathcal{D}$ must be of size (n, d) where d is the output dimension of the distribution (e.g., d=2 if π defines a 2-dimensional Gaussian pdf), and n is the number of i.i.d. samples from that distribution. For a python model, $\mathcal{D}$ must be a one-dimensional vector.

The following table lists the user-defined attributes of the class `Model`.

`Model` Class Inputs		
Attribute	**Type**	**Comment**
`model_type`	*str*	required, 'pdf' or 'python'
`n_params`	*int*	required
`model_name`	*str*	required if `model_type`='pdf'
`model_script`	*str*	required if `model_type`='python'
`error_covariance`	*float/ndarray*	default is 1
`prior_name`	*str/list of str*	prior used only in Bayesian inference
`prior_params`	*list/ndarray*	
`prior_copula`	*str*	
`prior_copula_params`	*list*	
`fixed_params`	*list*	

Input Attributes used by both types of models:

- `n_params`:
  `n_params` is the number of parameters in the model to be inferred, it is a required input of the class.

- `prior_name`, `prior_params`, `prior_copula`, `prior_copula_params`:
  In a Bayesian analysis, a prior for the parameters θ should be defined, which is done by calling `Distribution(dist_name=prior_name, copula=prior_copula)`. This build Distribution must have a `log_pdf` or a `pdf` method, which are evaluated using input parameters `prior_params`, `prior_copula_params`.

- `fixed_params`:
  The model can also take in as input a vector of fixed parameters, which are not being learnt. In this context, the model is fully parameterized by the vector $\begin{Bmatrix} \theta \\ \texttt{fixed\_params} \end{Bmatrix}$, where θ is being learnt during inference (the fixed parameters are appended at the end of the full parameter vector given as an input to the function that computes the data).

Input Attributes specific to distribution models:

- `model_name`:
  A probability model will be defined by calling `Distribution(dist_name=model_name)`, `model_name` can thus be a string that defines a distribution supported within UQpy, or a user-defined distribution. This distribution must have either a `log_pdf` method (preferred), or a `pdf` method. Very

importantly, these methods should be functions that accept exactly two inputs: `x` the point where to compute the pdf/log pdf, and `params` the value of the parameter vector characterizing that distribution. This means for instance that if one wants to define a distribution with a copula and copula parameters, they must define a custom distribution that is parameterized by a single parameter vector that concatenates the parameters of the marginals and the parameters of the copula into a single vector `params` (an example is provided in the file 'bivariate_normal_gumbel.py').

Input Attributes specific to python models:

- `model_script`:
  For a model defined using RunModel, `model_script` points to the '.py' file that computes $\mathcal{D}$, given as input a parameter vector θ (input `samples` of the function defined in `model_script`).

- `error_covariance`:
  The error term is assumed to have zero-mean and a known fixed covariance, given by `error_covariance`. `error_covariance` can be a scalar (then data points are i.i.d.) or a full covariance; default is 1.

- Inputs to RunModel:
  Class `Model` also accepts various input attributes which relate to the definition of the model in the RunModel module, namely, `model_object_name, input_template, var_names, output_script, output_object_name, ntasks, cores_per_task, nodes, resume, verbose, model_dir, cluster`.

- `model_name`:
  This input is not required for a python model, but useful when performing model selection for instance. If this input is None, the model name is built by concatenating the input `model_script` and `model_object_name`.

The following table describes the output attributes and methods of class `Model`.

`Model` Class Output Attributes and Methods	
Attribute/Method	**Type**
`log_like`	*function*
`prior`	instance of class `Distribution`

5.3.2 `UQpy.Inference.MLEstimation`

Computes the maximum likelihood estimator $\hat{\theta}$ of the model, i.e.

$$\hat{\theta} = argmax_{\Theta} \quad p(\mathcal{D}|\theta)$$

For a probabilistic model of the form $\mathcal{D} = h(\theta) + \epsilon$, $\epsilon \sim N(0, \sigma)$ with σ fixed and known and independent measurements $\mathcal{D}_i$, maximizing the likelihood is mathematically equivalent to minimizing the sum of squared residuals $\sum_i \left(\mathcal{D}_i - h(\theta)\right)^2$.

When the model is a probability model that possesses a `fit` method (see Distribution module), this fit method is used to compute the maximum likelihood parameters. Otherwise, i.e., for python models or distribution models without existing fit methods (custom distribution or distributions with copula for instance), a numerical optimization procedure is performed using the scipy.optimize.minimize module.

The following table summarizes the input attributes of the MLEstimation class.

`MLEstimation` Class Inputs		
Attribute	**Type**	**Comment**
`model`	instance of class `Model`	required
`data`	*ndarray*	required
`method_optim`	*string*	see input `method` of scipy.optimize.minimize
`x0`	*ndarray*	see scipy.optimize.minimize
`bounds`	*list*	see scipy.optimize.minimize
`iter_optim`	*int*	

More details on these input attributes are provided in the following.

- `model`:
  Model for which to performed inference, should be an instance of class Model.

- `data`:
  Data $\mathcal{D}$ used to perform inference, see section 5.3.1 for details on the size of the data matrix.

- `method_optim`, `x0`, `bounds`:
  These inputs are only used when a maximization of the log likelihood is performed using scipy.optimize.minimize (not a fit method), and determine some properties of the maximization procedure. The refer to

inputs `method`, `x0` and `bounds` of the scipy.optimize.minimize module, respectively.

- `iter_optim`: `iter_optim` defines the number of times the optimization procedure is run, with random initial guesses (it ignores `x0` in this case). The random initial guesses are sampled from the bounds provided by the user (input `bounds`), or between $[0, 1]$ if no bounds are provided. The identified maximum likelihood parameter vector is the one that yields the maximum log likelihood over all `iter_optim` runs of the maximization procedure.

The class returns two outputs attributes, the maximum likelihood estimate of the parameter vector $\hat{\theta}$ and the corresponding value of the log likelihood $\ln p(\mathcal{D}|\hat{\theta})$.

`MLEstimation` Class Output Attributes	
Attribute	**Type**
`param`	*ndarray*
`max_log_like`	*float*

Examples:
An example illustrating the use of the `MLEstimation` class is provided in the Maximum_Likelihood_Example.ipynb Jupyter script. Three different models are studied:

- a probability model with an existing fit method,
- a probability model without a fit method (custom distribution or distribution with copulas), which thus requires numerical optimization for maximum likelihood estimation,
- a python model defined with RunModel (a regression model).

5.3.3 `UQpy.Inference.BayesParameterEstimation`

Given some data $\mathcal{D}$, draws samples from the posterior pdf using Markov Chain Monte Carlo or Importance Sampling. Via Bayes theorem, the posterior pdf is as follows:

$$p(\theta|\mathcal{D}) = \frac{p(\mathcal{D}|\theta)p(\theta)}{p(\mathcal{D})}$$

Note that if no prior is defined in the model, the prior pdf is chosen as uninformative, i.e., $p(\theta) = 1$. UQpy also provides a diagnostics function, see

`Utilities` module, which performs some diagnostics on the outputs of the MCMC and IS procedures.

The code in `BayesParameterEstimation` simply defines a log_posterior function that evaluates $\tilde{p} = p(\mathcal{D}|\theta)p(\theta) \propto p(\theta|\mathcal{D})$. This function is then provided as the `log_pdf_target` input of the `MCMC` or `IS` classes.

Outputs of the class `BayesParameterEstimation` are samples from the posterior pdf (weighted samples in the case of IS, if one requires a set of unweighted samples to represent the posterior pdf, one can use the `resample` function provided in the `Utilities` module).

The following table summarizes the input attributes of the `BayesParameterEstimation` class.

`BayesParameterEstimation` Class Inputs		
Attribute	**Type**	**Comment**
`model`	instance of class `Model`	required
`data`	*ndarray*	required
`sampling_method`	*string*	required, 'MCMC' or 'IS'
`nsamples`	*int*	
`pdf_proposal`	*string/list*	only for IS
`pdf_proposal_params`	*list*	only for IS
`pdf_proposal_type`	*string/list*	only for MCMC
`pdf_proposal_scale`	*float/list*	only for MCMC
`algorithm`	*string*	only for MCMC
`jump`	*int*	only for MCMC
`nburn`	*int*	only for MCMC
`seed`	*ndarray*	only for MCMC if *None*, run ML estimation

More detailed explanations about each input attribute are as follows:

- `model`:
  Model for which to performed inference, should be an instance of class Model.

- `data`:
  Data $\mathcal{D}$ used to perform inference, see section 5.3.1 for details on the size of the data matrix.

- `sampling_method`:
  'MCMC'(default) to samples from the posterior via Markov Chain Monte Carlo or 'IS' to perform estimation via Importance Sampling.

- `nsamples`:
  Number of generated samples (weighted if IS) from the posterior.
- `pdf_proposal`, `pdf_proposal_params`:
  Used only if `sampling_method` is 'IS'. These inputs define the proposal distribution to sample from in Importance Sampling (see `IS` class in the `SamplingMethods` module). If no proposal distribution is provided, the algorithm samples from the prior defined for the model. Either a proposal distribution or a prior must be provided.
- `pdf_proposal_type`, `pdf_proposal_scale`, `nburn`, `jump`, `algorithm`, `seed`:
  Used only if `sampling_method` is 'MCMC'. These inputs define the inputs to MCMC, see `MCMC` class in the `SamplingMethods` module. If no seed is given, maximum likelihood is first performed and the maximum likelihood estimate of the parameter vector is used as the seed for MCMC.

The following table summarizes the output attributes of the `BayesParameterEstimation` class. See the `MCMC` and `IS` classes in the `SampleMethods` module for details.

`BayesParameterEstimation` Class Output Attributes		
Attribute	**Type**	**Comment**
`samples`	*ndarray*, size $(nsamples \times dim(\theta))$	
`weights`	*ndarray*, size $(nsamples,)$	only for IS
`accept_ratio`	*float*	only for MCMC

Examples:
Examples illustrating the use of the `BayesParameterEstimation` class are provided in the following Jupyter scripts:

- Bayesian_parameter_estimation_MCMC.ipynb
- Bayesian_parameter_estimation_IS.ipynb

These scripts illustrate Bayesian parameter estimation using MCMC and IS, respectively, for two different models:

- a probability model (Gaussian pdf, learn the posterior pdfs of its mean and variance from data),
- a python model defined with RunModel (regression model of the form $h(\theta) = \theta_1 x + \theta_2 x^2$, learn the posterior pdf of θ from data).

The notebooks also illustrate how to use the diagnostics function to check both the MCMC and IS outputs.

More complex examples of Inference for parameter estimation: A more complex example illustrating the use of the Inference module for parameter estimation is provided in the Parameter estimation - material homogenization.ipynb Jupyter script. This example consists in learning the material parameters, Young modulus and Poisson ratio, of the two materials composing a composite microstructure (matrix and fibers), when data is assumed to be measured at the macro level from tensile tests on a specimen. In this example, the model consists in running two FE codes, one simulating the behavior of the macro specimen, the other the behavior of a representative element of the microstructure. The FE simulations require use of the package Sfepy, the example is inspired from one of the Sfepy examples ([4]). The notebook illustrates the use of the `Model`, `MLEstimation` and `BayesParameterEstimation` modules of UQpy.

5.3.4 `UQpy.Inference.InfoModelSelection`

Model selection refers to the task of selecting a statistical model from a set of candidate models, given some data. A good model is one that is capable of explaining the data well. Given models of same explanatory power, the simplest model should be chosen (Ockam razor). Several simple information theoretic criteria can be used to compute a model's quality and perform model selection ([3]). UQpy implements three criteria:

- Bayesian information criterion (BIC)

$$BIC = ln(n)k - 2ln(\hat{L})$$

- Akaike information criterion (AIC)

$$AIC = 2k - 2ln(\hat{L})$$

- Corrected formula for AIC (AICc), for small data sets

$$AICc = AIC + \frac{2k(k+1)}{n-k-1}$$

For all formula above, k is the number of parameters characterizing the model, $\hat{L}$ is the maximum value of the likelihood function and n the number of data points. The best model is the one that minimizes the criterion. All three formulas have a model fit term (find the model that minimizes the negative log likelihood) and a penalty term that increases as the number of model parameters (model complexity) increases. A probability can be defined for each model as $P(m_i) \propto exp\left(-\frac{\text{criterion}}{2}\right)$.

`InfoModelSelection` calls `MLEstimation` to perform maximum likelihood estimation for each model. Thus inputs to `MLEstimation` can also be provided to `InfoModelSelection`, as lists of length the number of models. The procedure yields several outputs as attributes of the class, such as the fitted maximum likelihood parameters for all models, corresponding log likelihood values, model probabilities and so on (see details below). These outputs are given as lists, either sorted in the order they were given in the input `candidate_models` (if input `sorted_outputs` is set to *False*), or sorted in descending value of the model probabilities (default).

The following table provides a list of the inpiut attributes of that class.

`InfoModelSelection` Class Inputs		
Attribute/Method	**Type**	**Comment**
`candidate_models`	*list of models*	required
`data`	*ndarray*	required
`method`	*string*	default 'AIC'
`sorted_outputs`	*boolean*	default *True*
`x0`	*list* of length len(`candidate_models`)	inputs of `MLEstimation` class for each model
`iter_optim`		
`bounds`		
`method_optim`		

The following points provide some explanations about these input parameters:

- `candidate_models`:
  The list of candidate models, each of them must be an instance of class `Model`.

- `data`:
  Data $\mathcal{D}$ used to perform inference, see section 5.3.1 for details on the size of the data matrix.

- `method`:
  Criteria used for model selection: 'AIC' (default), 'BIC' or 'AICc'.

- `sorted_outputs`:
  If set to *True* (default), the outputs are returned as lists ordered by decreasing values of the model probabilities. If set to *False*, the outputs are returned as lists ordered in the same way as in `candidate_models`.

- `x0`, `iter_optim`, `bounds`, `method_optim`:
  Inputs to the `MLEstimation` class, see corresponding section. These inputs should be given as lists or length the number of models, ordered in the say way as `candidate_models`.

The following table provides a summary of the outputs attributes of the class `InfoModelSelection`.

`InfoModelSelection` Class Output Attributes	
Attribute	**Type**
`models`	*list of models*
`model_names`	*list of strings*
`fitted_params`	*list of ndarrays*
`criteria`	*list of floats*
`penalty_terms`	*list of floats*
`probabilities`	*list of floats*

The following points provide details about the outputs attributes of the class `InfoModelSelection`. All these outputs are lists of length the number of models, either ordered in the same way as the input list `candidate_models`, or in order of decreasing model probabilities.

- `models`:
  Instances of class models, same as `candidate_models` but possibly ordered in a different way.

- `model_name`:
  Names of the models.

- `fitted_params`:
  Maximum likelihood estimate of the parameter vector, for all models.

- `criteria`:
  Value of the criterion chosen for model selection, see formula in the theory section above.

- `penalty_terms`: Each criterion can be written as $criterion = -2ln(\hat{L}) +$ penalty_term, where the first term $-2ln(\hat{L})$ is a data-fit term, while the penalty term penalizes against complex models. Observing the penalty terms allows the user to understand if a model is chosen because it fits the data better than other models, or if it fits the data in the same way than competing models but is somehow less complex and thus preferred according to Ockam razor.

- `probabilities`:
  Models probabilities based on data, computed as $P(m_i) \propto exp\left(-\frac{\text{criterion}}{2}\right)$ for each model m_i

Examples:
An example illustrating the use of the `InfoModelSelection` class is provided in the Model_selection_info_criteria.ipynb Jupyter script. Two different examples are studied:

- selection between three univariate probability models,
- selection between three python models (polynomial regression models of different orders).

5.3.5 `UQpy.Inference.BayesModelSelection`

In the Bayesian approach to model selection, the posterior probability of each model is computed as:

$$P(m_i|\mathcal{D}) = \frac{p(\mathcal{D}|m_i)P(m_i)}{\sum_j p(\mathcal{D}|m_j)P(m_j)}$$

where the evidence (also called marginal likelihood) $p(\mathcal{D}|m_i)$ involves an integration over the parameter space:

$$p(\mathcal{D}|m_i) = \int_{\Theta} p(\mathcal{D}|m_i, \theta)p(\theta|m_i)d\theta$$

Currently, calculation of the evidence is performed using the method of the harmonic mean ([1]):

$$p(\mathcal{D}|m_i) = \left[\frac{1}{B}\sum_{b=1}^{B}\frac{1}{p(\mathcal{D}|m_i, \theta_b)}\right]^{-1}$$

where $\theta_{1,\cdots,B}$ are samples from the posterior pdf of θ. In UQpy, these samples are obtained by running `BayesParameterEstimation` using MCMC. However, note that this method is known to yield evidence estimates with large variance. Future releases of UQpy will include more robust methods for computation of model evidences. Also, it is known that results of such Bayesian model selection procedure usually highly depends on the choice of prior for the parameters of the competing models, thus the user should carefully define such priors when creating instances of the `Model` class.

Similarly to the `InfoModelSelection` class, the `BayesModelSelection` class takes as inputs the data, candidate models, along with additional inputs that are lists of length the number of models and define inputs to the MCMC procedure for all models. Additionally, `BayesModelSelection` takes as input the prior probabilities of the models. The procedure yields outputs such as posterior model probabilities, evidence etc. as lists, either sorted in the same order as given in `candidate_models` or sorted by decreasing model probabilities.

`BayesModelSelection` Class Inputs		
Attribute/Method	**Type**	**Comment**
`candidate_models`	*list of models*	required
`data`	*ndarray*	required
`prior_probabilities`	*ndarray*	default $\frac{1}{M}$ for all M models
`sorted_outputs`	*boolean*	default *True*
`n_samples`	*lists* of length the number of candidate models	inputs of class `BayesParameterEstimation` (uses MCMC)
`pdf_proposal_type`		
`pdf_proposal_scale`		
`algorithm`		
`jump`		
`nburn`		
`seed`		

The following points provide some explanations about these input parameters:

- `candidate_models`:
  The list of candidate models, each of them must be an instance of class `Model`.

- `data`:
  Data $\mathcal{D}$ used to perform inference, see section 5.3.1 for details on the size of the data matrix.

- `prior_probabilities`:
  Prior model probabilities $P(m_i)$ as a *list of floats* or *ndarray*, default is a list of $\frac{1}{M}$ for all M models.

- `sorted_outputs`:
  If set to *True* (default), the outputs are returned as lists ordered by decreasing values of the model probabilities. If set to *False*, the outputs are returned as lists ordered in the same way as in `candidate_models`.

- `pdf_proposal_type`, `pdf_proposal_scale`, `algorithm`, `jump`, `nburn`, `seed`:
  Inputs to the `BayesParameterEstimation` class, see corresponding section. These inputs should be given as lists or length the number of models, ordered in the say way as `candidate_models`.

The following table provides a summary of the outputs attributes of the class `BayesModelSelection`.

`BayesModelSelection` Class Output Attributes	
Attribute	**Type**
`models`	*list of models*
`model_names`	*list of strings*
`evidences`	*list of floats*
`mcmc_outputs`	*list of instances of* `BayesParameterEstimation`
`probabilities`	*list of floats*

The following points provide details about the outputs attributes of the class `BayesModelSelection`. All these outputs are lists of length the number of models, either ordered in the same way as the input list `candidate_models`, or in order of decreasing model probabilities.

- `models`:
  Instances of class models, same as `candidate_models` but possibly ordered in a different way.

- `model_names`:
  Names of the models.

- `evidences`:
  Value of the evidence $p(\mathcal{D}|m_i)$ for each model m_i.

- `mcmc_outputs`: Objects of the class `BayesParameterEstimation`, which have as attributes both the samples of the posterior pdf for all models and the acceptance ratio of the chains. See section on `BayesParameterEstimation`.

- `probabilities`:
  Value of the posterior probability $P(m_i|\mathcal{D})$ for each model m_i.

Examples:
An example illustrating the use of the `BayesModelSelection` class is provided in the Bayesian model selection.ipynb Jupyter script. The example studied is the selection between three python models (polynomial regression models of different orders). Gaussian priors are assumed for the parameters, rendering the problem tractable, meaning that the true posterior pdfs and values of the evidence for each model can be computed analytically. Analytical results are compared with outputs of the `BayesModelSelection` algorithm.

5.4 `Reliability` Module

Reliability of a structural system refers to the assessment of its failure (i.e the structure no longer satisfies some performance measures), given the model uncertainty in the structural, environmental and load parameters. Given a vector of random variables $\mathbf{X} = \{X_1, X_2, \ldots, X_n\} \in \mathcal{D}_{\mathbf{X}} \subset \mathbb{R}^n$, where $\mathcal{D}$ is the domain of interest and $f_{\mathbf{X}}(\mathbf{x})$ is its joint probability density function then, the probability that the system will fail is defined as

$$P_f = \mathbb{P}(g(\mathbf{X}) \leq 0) = \int_{D_f} f_{\mathbf{X}}(\mathbf{x})d\mathbf{x} = \int_{\{\mathbf{X}:g(\mathbf{X})\leq 0\}} f_{\mathbf{X}}(\mathbf{x})d\mathbf{x} \tag{2}$$

where $g(\mathbf{X})$ is the so-called limit-state function. Formulation of reliability methods in `UQpy` is made on the standard normal space $\mathbf{U} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_n)$ which means that a nonlinear isoprobabilistic transformation from the generally non-normal parameter space $\mathbf{X} \sim f_{\mathbf{X}}(\cdot)$ is required (see Section 5.7).

The `Reliability` module consists of classes and functions to provide simulation-based estimates of probability of failure from a given user-defined computational model and failure criterion. It is imported in a python script using the following command:

```python
from UQpy import Reliability
```

The `Reliability` module has the following classes, each corresponding to a method for probability of failure estimation:

Class	**Method**
`SubsetSimulation`	Subset Simulation
`TaylorSeries`	FORM/SORM

Each class can be imported individually into a python script. For example, the `SubsetSimulation` and the `TaylorSeries` classes can be imported to a script using the following commands:

```
from UQpy.SampleMethods import SubsetSimulation

from UQpy.SampleMethods import TaylorSeries
```

The following subsections describe each class, their respective inputs and attributes, and their use.

5.4.1 `UQpy.Reliability.SubsetSimulation`

In the subset simulation method the probability of failure P_f is approximated by a product of probabilities of more frequent events. That is, the failure event $G = \{\mathbf{u} \in \mathbb{R}^n : G(\mathbf{u}) \leq 0\}$, is expressed as the of union of M nested intermediate events $G_1, G_2, \cdots, G_M$ such that $G_1 \supset G_2 \supset \cdots \supset G_M$, and $G = \cap_{i=1}^{M} G_i$. The intermediate failure events are defined as $G_i = \{G(\mathbf{u}) \leq b_i\}$, where $b_1 > b_2 > \cdots > b_i = 0$ are positive thresholds selected such that each conditional probability $P(G_i|G_{i-1}),\ i = 2, 3, \cdots, M-1$ equals a target probability value p_0. The probability of failure P_f is estimated as:

$$P_f = P\left(\cap_{i=1}^{M} G_i\right) = P(F_1) \prod_{i=2}^{M} P(G_i|G_{i-1}) \tag{3}$$

where the probability $P(F_1)$ is computed through Monte Carlo simulations. In order to estimate the conditional probabilities $P(G_i|G_{i-1}),\ j = 2, 3, \cdots, M$ generation of Markov Chain Monte Carlo (MCMC) samples from the conditional pdf $p_{\mathbf{U}}(\mathbf{u}|G_{i-1})$ is required. In the context of subset simulation, the Markov chains are constructed through a two-step acceptance/rejection criterion. Starting from a Markov chain state $\mathbf{x}$ and a proposal distribution $q(\cdot|\mathbf{x})$, a candidate sample $\mathbf{y}$ is generated. In the first stage, the sample $\mathbf{y}$ is accepted/rejected with probability

$$\alpha = \min\left\{1, \frac{p(\mathbf{y})q(\mathbf{x}|\mathbf{y})}{p(\mathbf{x})q(\mathbf{y}|\mathbf{x})}\right\} \tag{4}$$

and in the second stage is accepted/rejected based on whether the sample belongs to the failure region G_j. Currently `UQpy` supports the Metropolis-Hastings (MH), the Component-wise Metropolis Hastings (MMH) and the affine invariant ensemble MCMC algorithm (see Section 5.2).

The `SubsetSimulation` class is imported using the following command:

```
from UQpy.Reliability import SubsetSimulation
```

The attributes of the `SubsetSimulation` class are listed below:

`SubsetSimulation` Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
`dimension`	Input		⋆
`nsamples_init`	Input		⋆
`nsamples_ss`	Input	⋆	
`p_cond`	Input		⋆
`algorithm`	Input		⋆
`pdf_target_type`	Input		⋆
`pdf_target`	Input	⋆	
`pdf_target_params`	Input		⋆
`pdf_proposal_type`	Input		⋆
`pdf_proposal_scale`	Input		⋆
`seed`	Input		⋆
`model_type`	Input		⋆
`model_script`	Input	⋆	
`input_script`	Input		⋆
`output_script`	Input		⋆
`samples`	Output		
`g`	Output		
`g_level`	Output		
`pf`	Output		

A brief description of each attribute can be found in the table below:

`SubsetSimulation` Class Attributes			
Attribute	**Type**	**Options**	**Default**
`dimension`	*integer*		`dimension` = 1
`samples_init`	*nparray*		`None`
`nsamples_ss`	*integer*		`None`
`p_cond`	*float*	0 < `p_cond` < 1	`p_cond` = 0.1
`algorithm`	*string*	'MMH' 'Stretch'	'MMH'
`pdf_target_type`	*string*	'marginal_pdf' 'joint_pdf'	'marginal_pdf'
`pdf_target`	*function* *string*		Normal($\mathbf{0}, \mathbf{I}$)
`pdf_target_params`	*float* *float list*		`None`
`pdf_proposal_type`	*string*	'Normal' 'Uniform'	'Uniform'
`pdf_proposal_scale`	*float* *float list*		`algorithm` = 'MMH' or 'MH' [1,1,...,1] `algorithm`='Stretch' 2
`model_type`	*string*	See `UQpy.RunModel`	See `UQpy.RunModel`
`model_script`	*string*	See `UQpy.RunModel`	See `UQpy.RunModel`
`input_script`	*string*	See `UQpy.RunModel`	See `UQpy.RunModel`
`output_script`	*string*	See `UQpy.RunModel`	See `UQpy.RunModel`
`samples`	*nparray list*		
`g`	*nparray list*		
`g_level`	*list*		
`pf`	*float*		

Detailed Description of `SubsetSimulation` Class Attributes:

Input Attributes:

- `dimension`:
  A scalar integer value defining the dimension of the random variables.

- `samples_init`
  Specifies the initial samples for subset/level 0. The size of the array `samples_init` must be `nsamples_ss`×`dimension`. These samples can be generated in any way the user chooses.

  If `samples_init` is not specified, the subset/level 0 samples are drawn internally in `SubsetSimulation` using the component-wise Modified Metropolis-Hastings algorithm.

- `nsamples_ss`
  Specifies the number of samples to be generated in each conditional level (i.e. per subset). `nsamples_ss` must be specified. There is no default value.

- `p_cond`
  Specifies the conditional probability for each subset.

  The current implementation does not allow for variable conditional probabilities (i.e. setting different conditional probabilities for each level).

  The current implementation does not allow for the conditional probabilities to be defined implicitly by instead specifying the intermediate failure domains explicitly.

- `algorithm`:
  Specifies the MCMC algorithm used to generate samples in each conditional level. `SubsetSimulation` currently supports two commonly-used algorithms.

  - 'MMH':
    Component-wise modified Metropolis-Hastings algorithm. For a description of the algorithm, see [2].
  - 'Stretch':
    Affine invariant ensemble sampler employing "stretch" moves. For a description of the algorithm, see [7].

  `SubsetSimulation` currently does not support the conventional Metropolis-Hastings algorithm.

- `pdf_target_type`:
  This is used for Markov Chain Monte Carlo (MCMC) sampling from the conditional probability densities in subset simulation. For details, the user is referred to documentation for `UQpy.SampleMethods.MCMC` in Section 5.2.5

- `pdf_target`:
  This is used for Markov Chain Monte Carlo (MCMC) sampling from the conditional probability densities in subset simulation. For details,

the user is referred to documentation for `UQpy.SampleMethods.MCMC` in Section 5.2.5

- `pdf_target_params`:
  This is used for Markov Chain Monte Carlo (MCMC) sampling from the conditional probability densities in subset simulation. For details, the user is referred to documentation for `UQpy.SampleMethods.MCMC` in Section 5.2.5

- `pdf_proposal_type`:
  This is used for Markov Chain Monte Carlo (MCMC) sampling from the conditional probability densities in subset simulation. For details, the user is referred to documentation for `UQpy.SampleMethods.MCMC` in Section 5.2.5

- `pdf_proposal_scale`:
  This is used for Markov Chain Monte Carlo (MCMC) sampling from the conditional probability densities in subset simulation. For details, the user is referred to documentation for `UQpy.SampleMethods.MCMC` in Section 5.2.5

- `model_type`
  This is used to evaluate the model at each sample point using the `RunModel` class. For details, the user is referred to documentation for `UQpy.RunModel` in Section 5.1.

- `model_script`
  This is used to evaluate the model at each sample point using the `RunModel` class. For details, the user is referred to documentation for `UQpy.RunModel` in Section 5.1.

  Note that a computational model must be specified using `model_script`. Without this model, `SubsetSimulation` cannot run.

- `input_script`
  This is used to evaluate the model at each sample point using the `RunModel` class. For details, the user is referred to documentation for `UQpy.RunModel` in Section 5.1.

- `output_script`
  This is used to evaluate the model at each sample point using the

`RunModel` class. For details, the user is referred to documentation for `UQpy.RunModel` in Section 5.1.

Output Attributes:

- `samples`:
  Contains the sample values from each conditional level as a list of numpy arrays.

  Each item of the list is a numpy array containing the samples from the corresponding conditional level. For example, `SubsetSimulation.samples[0]` contains a numpy array of dimension `nsamples_ss`×`dimension` with the samples from conditional level 0 (i.e. the initial sample set).

- `g`
  Returns the scalar values of the performance function evaluated by the computational model at each point in `samples`. `g` is structured in the same manner as `samples` (a *numpy array list*) with each entry equal to the performance function evaluation of the corresponding sample.

  By convention, failure of a given sample `sample[i][j]` is defined by `g[i][j]` < 0, where `i` indexes the conditional level and `j` indexes the sample number. For use with `SubsetSimulation`, the user's computational model must return a scalar value that follows this convention. The value is passed from `RunModel` into `SubsetSimulation` through the attribute `RunModel.model_eval.QOI` as detailed in Section 5.1.

- `g_level`
  Specifies the value of the performance function for each conditional level. `g_level` is structured as a list with each entry of the list equal to the value of the corresponding performance function at the respective conditional level. For example, `g_level[3]` corresponds to the performance function value that defines the third subset.

  Note that `g_level` is implicitly defined by the samples and `p_cond`. `UQpy` currently does not support the direct assignment of conditional performance levels.

- `pf`
  Probability of failure estimate from subset simulation

`SubsetSimulation` **Examples:**
Two examples illustrating the use of the `MCMC` class are provided in the following Jupyter scripts.

- MCMC_Example1.ipynb:
  In this example, the three MCMC algorithms are used to generate 1000 samples from a two-dimensional Rosenbrock pdf. The Rosenbrock pdf is defined as a function directly in the script.

- MCMC_Example2.ipynb:
  In this example, the three MCMC algorithms are used to generate 1000 samples from a two-dimensional Rosenbrock pdf. The Rosenbrock pdf is defined as a function in the 'custom_pdf.py' script.

5.4.2 `UQpy.Reliability.TaylorSeries`

These reliability methods utilize a Taylor series expansion to approximate the performance function $g(\mathbf{X})$ locally at a design point by simplifying $f_{\mathbf{X}}(\mathbf{x})$ and thus, enhancing the solution of the integral in Eq.(2). In this category belong the First Order Reliability Method (FORM) and the Second Order Reliability Method (SORM). In the context of FORM the performance function is linearized according to

$$G(\mathbf{U}) \approx G(\mathbf{U}^{\star}) + \nabla G_{|\mathbf{U}^{\star}}(\mathbf{U}-\mathbf{U}^{\star})^{\intercal} \tag{5}$$

where $\mathbf{U}^{\star}$ is expansion point, $G(\mathbf{U})$ is the performance function evaluated in the standard normal space and $\nabla G_{|\mathbf{U}^{\star}}$ is the gradient of $G(\mathbf{U})$ evaluated at $\mathbf{U}^{\star}$. The probability failure can be calculated by

$$P_{f,\text{form}} = \Phi(-\beta_{HL}) \tag{6}$$

where $\Phi(\cdot)$ is the standard normal cumulative distribution function and $\beta_{HL} = ||\mathbf{U}^{*}||$ is the norm of the design point known as Hasofer-Lind reliability index calculated with the Hasofer-Lind-Rackwitz-Fiessler (HLRF) algorithm. In SORM the performance function is approximated by a second-order Taylor series around the design point according to

$$G(\mathbf{U}) = G(\mathbf{U}^{\star}) + \nabla G_{|\mathbf{U}^{\star}}(\mathbf{U}-\mathbf{U}^{\star})^{\intercal} + \frac{1}{2}(\mathbf{U}-\mathbf{U}^{\star})\mathbf{H}(\mathbf{U}-\mathbf{U}^{\star}) \tag{7}$$

where $\mathbf{H}$ is the Hessian matrix of the second derivatives of $G(\mathbf{U})$ evaluated at $\mathbf{U}^{*}$. After the design point $\mathbf{U}^{*}$ is identified and the probability of failure

$P_{f,\text{form}}$ is calculated with FORM a correction is made according to

$$P_{f,\text{sorm}} = \Phi(-\beta_{HL}) \prod_{i=1}^{n-1} (1 + \beta_{HL}\kappa_i)^{-\frac{1}{2}} \tag{8}$$

where κ_i is the $i-th$ curvature.

The `TaylorSeries` class is imported using the following command:

```
from UQpy.Reliability import TaylorSeries
```

The attributes of the `TaylorSeries` class are listed below:

`TaylorSeries` Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
`dimension`	Input	⋆	
`dist_name`	Input see `UQpy.Distribution` class	⋆	
`dist_params`	Input see `UQpy.Distribution` class	⋆	
`n_iter`	Input		⋆
`corr`	Input		⋆
`method`	Input	⋆	
`algorithm`	Input	⋆	
`seed`	Input		⋆
`model_script`, `model_object_name`, `input_template`, `var_names`, `output_script`, `ntasks`, `cores_per_task`, `resume`, `output_object_name`	Input see `UQpy.RunModel` class		
`DesignPoint_X`	Output		
`DesignPoint_U`	Output		
`Prob_FORM`	Output		
`Prob_SORM`	Output		
`HL_beta`	Output		
`iterations`	Output		

A brief description of each attribute can be found in the table below:

TaylorSeries Class Attributes			
Attribute	**Type**	**Options**	**Default**
`dimension`	*integer*		None
`dist_name`	see `UQpy.Distribution` class		None
`dist_params`	see `UQpy.Distribution` class		None
`corr`	see `UQpy.Transformation` class		`np.eye(dimension)`
`method`	*string*	'FORM' 'SORM'	None
`n_iter`	*integer*	`n_iter` > 0	`n_iter` $= 1000$
`algorithm`	*string*	'HL' '(Hasofer-Lind)'	None
`seed`	*ndarray*		`np.zeros((1, dimension))`
`model_script`, `model_object_name`, `input_template`, `var_names`, `output_script`, `ntasks`, `cores_per_task`, `resume`, `output_object_name`	see `UQpy.RunModel` class	see `UQpy.RunModel` class	see `UQpy.RunModel` class
`DesignPoint_X`	*ndarray*		
`DesignPoint_U`	*ndarray*		
`Prob_FORM`	*float*		
`Prob_SORM`	*float*		
`HL_beta`	*float*		
`iterations`	*integer*		

Detailed Description of `TaylorSeries` Class Attributes:

Input Attributes:

- `dimension`:
  A scalar integer value defining the dimension of the random variables.

- `dist_name`
  Specifies the probability distribution model for each random variable. Details about this attribute can be found in `UQpy.Distribution`.

- `dist_params`
  Specifies the parameters for each probability model. Details about this attribute can be found in `UQpy.Distribution`.

- `corr`
  Specifies the correlation structure of the random vector. If not defined, we assume independent random variables.

$$\texttt{corr} = \begin{bmatrix} 1.0 & 0.0 & \dots & 0.0 \\ 0.0 & 1.0 & \dots & 0.0 \\ \vdots & \vdots & \ddots & \vdots \\ 0.0 & 0.0 & \dots & 1.0 \end{bmatrix}$$

Details about this attribute can be found in `UQpy.Transformation`.

- `method`:
  Specifies the method from the family of Taylor Series expansion. `TaylorSeries` supports two commonly-used algorithms.
  - 'FORM':
    First Order Reliability Method.
  - 'SORM':
    Second Order Reliability Method.
- `n_iter`:
  Maximum number of iterations of the Hasofer-Lind iterative method.
- `algorithm`:
  Specifies the algorithm used to solve the optimization problem for finding the design point. `TaylorSeries` currently supports the **Hasofer-Lind** method.
- `seed`:
  Specifies the initial point in the original parameter space (not in the standard normal space) of the search algorithm in the Hasofer-Lind method.

Output Attributes:

- `DesignPoint_X`:
  Design point in the original parameter space.
- `DesignPoint_U`
  Design point in the standard normal space.
- `Prob_FORM`
  Probability of failure obtained with FORM.

- `Prob_FORM`
  Probability of failure calculated with SORM (if `method`='SORM').
- `HL_beta`
  Hasofer-Lind reliability index.
- `iterations`
  Total number of function calls.

`TaylorSeries` **Examples:**
An examples illustrating the use of the `TaylorSeries` class is provided in the following Jupyter scripts.

- TaylorSeries_Example1.ipynb:
  This benchmark case is a simple structural reliability problem defined in a two-dimensional parameter space consisting of a resistance R and a stress S. The failure happens when the stress is higher than the resistance, leading to the following limit-state function:

$$g(\mathbf{X}) = R - S \tag{9}$$

  where $\mathbf{X} = \{R, S\}$. The two random variables are independent and distributed according to the following normal distributions: $R \sim N(5, 0.8)$ and $S \sim N(2, 0.6)$.

5.5 `Surrogates` Module

The `Surrogates` module consists of classes and functions to build simplified mathematical expressions to interpolate data and serve as a meta-model, surrogate model, or emulator. It is imported in a python script using the following command:

```
from UQpy import Surrogates
```

The `Surrogates` module has the following classes, each corresponding to a different surrogate model form:

Class	**Method**
`SROM`	Stochastic Reduced Order Model
`Krig`	Kriging

5.5.1 `UQpy.Surrogates.SROM`

Theory

`SROM` takes a set of samples and attributes of a distribution and optimizes the sample probability weights according to the method of Stochastic Reduced Order Models as defined by Grigoriu [8]. This method identifies the weights associated with samples, such that total error between distribution, moments and correlation of random variables is minimized. This method is explained in detail in Grigoriu [8].

Using the `SROM` Class

The SROM class is imported using the following command:

```
from UQpy.Surrogates import SROM
```

The attributes of the `SROM` class are listed below:

`SROM` Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
`samples`	Input	⋆	
`cdf_target`	Input	⋆	
`cdf_target_params`	Input	⋆	
`properties`	Input		⋆
`moments`	Input	⋆	
`correlation`	Input		⋆
`weights_error`	Input		⋆
`weights_distribution`	Input		⋆
`weights_moments`	Input		⋆
`weights_correlation`	Input		⋆
`sample_weights`	Output		

A brief description of each attribute can be found in the table below:

SROM Class Attributes			
Attribute	**Type**	**Options**	**Default**
samples	*ndarray*		None
cdf_target	*function/string list*		None
cdf_target_params	*ndarray list*		None
properties	*boolean list*	True False	[True,True,True,False]
moments	*ndarray list*		None
correlation	*ndarray*		Identity matrix
weights_error	*list*		[1, 0.2, 0]
weights_distribution	*ndarray list*		Array of ones with size of samples
weights_moments	*ndarray list*		$\frac{1}{\texttt{moments}^2}$
weights_correlation	*ndarray list*		
sample_weights	*ndarray*		

Detailed Description of SROM Class Attributes:

Input Attributes:

- `samples`:
  An array or list containing the samples from which to build the Stochastic Reduced Order Model.

- `cdf_target`:
  A list of functions or strings specifying the Cumulative Distribution Functions (CDFs) of the random variables.

  If `cdf_target[i]` is a string, the distribution is matched with its corresponding cdf (`cdf`) in the `Distributions` module (see Sec. 6.1) or the cdf defined by 'custom_dist.py' (again see Sec. 6.1).

  if `cdf_target[i]` is a function, it must be defined in the user's Python script and passed directly as a function.

  `cdf_target` can contain an arbitrary combination of strings and functions.

  When `dimension` > 1, `cdf_target` may be specified as a string/function or a list of strings/functions assigned to each dimension. When specified as a string/function, the same cdf is specified for all dimensions.

- `cdf_target_params`:
  A list of parameters corresponding to each random variable where the parameters for each random variable are assigned as a numpy array.

  Example: `cdf_target` = [‘Gamma’] and `cdf_target_params` = $[np.array([2, 1, 3])]$, where the random variables have gamma distribution with shape, shift and scale parameters equal to 2, 1 and 3 respectively.

- `properties`:
  A boolean list specifying which properties of the distribution are to be included in the objective function. The list is of size 4 with the items of the list defined as follows:

  1. it CDF: Minimize error in the match to the cumulative distribution function.
  2. it mean: Minimize error in the first-order moments about the origin.
  3. *variance*: Minimize error in the second-order moments about the origin.
  4. *correlation*: Minimize error in correlation.

  ‘True’ includes the corresponding property in the objection function and ‘False’ excludes it.

- `moments`:
  A list of numpy arrays specifying the first and second-order moments about the origin for each random variable. `SROM` supports the following size of `moments` array:

  – Array of size 1 × `dimension`: If error in either, but not both, first or second-order moments is included in SROM.
  – Array of size 2 × `dimension`: If error in both first and second-order moments are included in the SROM. The first row contains first-order moments and the second row contains the second-order moments.

- `correlation`:
  An array specifying the correlations among the random variables. It is defined such that size of array is `dimension` × `dimension`.

- `weights_error`:
  `SROM` generates `sample_weights` which minimize the error between the cdf, moments, and correlation of the samples and the probability model. `weights_error` specifies weights assigned to each property in the objective function as outlined in [8]. It is a list of size 3 with the items defined as follows:

  - *Item 1*: Weight assigned to the cumulative distribution function.
  - *Item 2*: Weight assigned to the first and second marginal moments.
  - *Item 3*: Weight assigned to the correlation matrix.

  Default values are set as in [8].

- `weights_distribution`:
  A list of arrays containing weights defining the error in distribution at each sample of the random variables. `SROM` supports the following options for `weights_distribution`:

  - `None`: Default value is defined as an array of the same size as `samples` with each value equal to 1. For default value, See [8].
  - Array of size $1 \times$ `dimension`: Equal weights are assigned to all samples in same dimension.
  - Arbitrary array of the same size as `samples`: User specifies all weights explicitly.

- `weights_moments`:
  A list of arrays containing weights defining the error in moments in each dimension. `SROM` supports the following options for `weights_moments`:

  - `None`: Default value is defined as array of the same size as `moments` with each value equal to the reciprocal of the square of `moments`. For default value, see [8].
  - Array of size $1 \times$ `dimension`: Equal weights are assigned to both moments in same dimension.
  - Array of size same as `moments`: User specifies all weights explicitly.

- `weights_correlation`:
  A list of arrays containing the weights defining the error in correlation among random variables. It is define such that the size of the array is the same as `correlation`. For default value, See [8].

Output Attributes:

- `sample_weights`:
  The generated SROM weights corresponding to `samples`. The samples are returned as a numpy array with each sampling having a corresponding weight.

Examples:
Two examples illustrating the use of the `SROM` class are provided in the following Jupyter scripts.

- SROM_Example1.ipynb:
  In this example, the `STS` is used to generate 16 samples from a two-dimensional Gamma pdf. The Gamma pdf is defined as a function directly in the script. Then, `SROM` is used to obtain sample weights.

- SROM_Example2.ipynb:
  In this example, sample weights are compared when `SROM` is called using default values for `weights_distribution` and `weights_moments` and when `SROM` is called with user-defined values for `weights_distribution` and `weights_moments`.

- SROM_Example3.ipynb:
  In this example, SROM is used to estimate the distribution of eigenvalues of a spring-mass system, where stiffness of spring is treated as a random variable, which follows gamma distribution. Distribution of eigenvalues obtained by SROM method is compared with the Monte Carlo estimate.

5.5.2 `UQpy.Surrogates.Krig`

Theory
`Krig` class defines an approximate surrogate model or response surface which can be used to predict function values at unknown location. Kriging gives the best unbiased linear predictor at the intermediate samples. `Krig` class generates a model $\hat{y}$ that express the response surface as a realization of regression model and gaussian random process.

$$\hat{y}(x) = \mathcal{F}(\beta, x) + z(x)$$

Regression model ($\mathcal{F}$) is linear combination of 'p' chosen scalar basis function.

$$\mathcal{F}(\beta, x) = \beta_1 f_1(x) + \cdots + \beta_p f_p(x) = f(x)^T \beta$$

The random process $z(x)$ have mean zero and covariance is defined through correlation matrix($\mathcal{R}(\theta, s, x)$), which depends on hyperparameters(θ) and samples(s).

$$E\big[z(s)z(x)\big] = \sigma^2 \mathcal{R}(\theta, s, x)$$

Hyperparameters are estimate by maximizing the log-likehood function.

$$\log(p(y|x,\theta)) = -\frac{1}{2}y^T\mathcal{R}^{-1}y - \frac{1}{2}\log(|\mathcal{R}|) - \frac{n}{2}\log(2\pi)$$

Once hyperparameters are computed, correlation matrix($\mathcal{R}$) and basis functions are evaluated at sample points(F). Then, correlation coefficient(β) and process variance(σ^2) can be computed using following equations.

$$(F^T R^{-1} F)\beta^* = F^T R^{-1} Y$$

$$\sigma^2 = \frac{1}{m}(Y - F\beta^*)^T R{-}1(Y - F\beta^*)$$

The final predictor function can be defined as:

$$\hat{y}(x) = f(x)^T\beta^* + r(x)^T R^{-1}(Y - F\beta^*)$$

Using the `Krig` Class

The Krig class is imported using the following command:

```
from UQpy.Surrogates import Krig
```

The attributes of the `Krig` class are listed below:

`Krig` Class Attribute Definitions

Attribute	**Input/Output**	**Required**	**Optional**
`samples`	Input	⋆	
`values`	Input	⋆	
`reg_model`	Input	⋆	
`corr_model`	Input	⋆	
`corr_model_params`	Input		⋆
`bounds`	Input		⋆
`op`	Input		⋆
`n_opt`	Input		⋆
`interpolate`	Output		
`jacobian`	Output		

A brief description of each attribute can be found in the table below:

`Krig` Class Attributes			
Attribute	**Type**	**Options**	**Default**
`samples`	*ndarray/list*		`None`
`values`	*ndarray/list*		`None`
`reg_model`	*function/string*	`Constant` `Linear` `Quadratic`	`None`
`corr_model`	*function/string*	`Exponential` `Gaussian` `Linear` `Cubic` `Spherical` `Spline`	`None`
`corr_model_params`	*ndarray*		`[1,1,...,1]`
`bounds`	*list*		$[10^{-3}, 10^{7}]$
`op`	*boolean*		True
`n_opt`	*int*		1
`interpolate`	*function*		
`jacobian`	*function*		

Detailed Description of `Krig` Class Attributes:

Input Attributes:

- `samples`:
  An array or list containing the samples from which to build the Kriging surrogate. Size of the array should be $m \times n$, where 'm' is number of samples and 'n' is dimension of sample space.

- `values`:
  An array or list of function values evaluated at the samples. Size of the array should be $m \times q$, where 'q' is dimension of output space.

- `reg_model`:
  A function or string defining the trend of the model, which defines the basis function. There are three predefined regression model inside the class i.e. 'Constant', 'Linear' and 'Quadratic' regression model.

Constant:

$$f_1(x) = 1 \qquad J_f = [O_{n\times 1}]$$

Linear:

$$f_1(x) = 1, \quad f_2(x) = x_1, \ \ldots, \quad f_{n+1}(x) = x_n$$

$$J_f = [O_{n\times 1} \quad I_{n\times n}]$$

Quadratic:

$$\begin{aligned}
&f_1(x) = 1 \\
&f_2(x) = x_1, \quad f_3(x) = x_2, \ \ldots, \quad f_{n+1}(x) = x_n \\
&f_{n+2}(x) = x_1^2, \quad f_{n+3}(x) = x_1x_2, \ \ldots, \quad f_{2n+1}(x) = x_1x_n \\
&f_{2n+2}(x) = x_2^2, \quad f_{n+3}(x) = x_2x_3, \ \ldots, \quad f_{3n}(x) = x_2x_n \\
&\ldots \quad \ldots f_{\frac{(n+1)(n+2)}{2}} = x_n^2
\end{aligned}$$

$$J_f = [O_{n\times 1} \quad I_{n\times n} \quad H]$$

where H can be illustrated as:

$$\begin{aligned}
n = 2 \quad &: \quad H = \begin{bmatrix} 2x_1 & x_2 & 0 \\ 0 & x_1 & 2x_2 \end{bmatrix} \\
n = 3 \quad &: \quad H = \begin{bmatrix} 2x_1 & x_2 & x_3 & 0 & 0 & 0 \\ 0 & x_1 & 0 & 2x_2 & x_3 & 0 \\ 0 & 0 & x_1 & 0 & x_2 & 2x_3 \end{bmatrix}
\end{aligned}$$

This class also support an user defined function.

```
def reg_model(x):
    ...
    return fx, jf
```

where, fx and jf are value of basis function and it's Jacobian at sample point ‘x’.

$$\text{fx} = \begin{bmatrix} f_1(x) & f_2(x) & ... & f_l(x) \end{bmatrix}$$

$$\text{jf} = \begin{bmatrix} \frac{\partial f_1(x)}{\partial x_1} & \frac{\partial f_2(x)}{\partial x_1} & \cdots & \frac{\partial f_l(x)}{\partial x_1} \\ \frac{\partial f_1(x)}{\partial x_2} & \frac{\partial f_2(x)}{\partial x_2} & \cdots & \frac{\partial f_l(x)}{\partial x_2} \\ . & & & \\ . & & & \\ \frac{\partial f_1(x)}{\partial x_n} & \frac{\partial f_2(x)}{\partial x_n} & \cdots & \frac{\partial f_l(x)}{\partial x_n} \end{bmatrix}$$

- `corr_model`:
  A function or string defining the correlation among the covariates of model. It explains the how similar are two points. There are six pre-defined correlation model inside the class i.e. ‘Exponential’, ‘Gaussian’, ‘Linear’, ‘Cubic’, ‘Spherical’ and ‘Spline’.

$$\mathcal{R}(\theta, s, x) = \prod_{j=1}^{n} \mathcal{R}_j(\theta, s_j - x_j)$$

Name	$\mathcal{R}_j(\theta, d_j)$
Exponential	$\exp(-\theta_j \lvert d_j \rvert)$
Gaussian	$\exp(-\theta_j d_j^2)$
Linear	$\max\{0, 1 - \theta_j \lvert d_j \rvert\}$
Spherical	$1 - 1.5\zeta_j + 0.5\zeta_j^3$
Cubic	$1 - 3\zeta_j^2 + 2\zeta_j^3$
Spline	$\xi(\zeta_j)(10), \quad \zeta_j = \lvert d_j \rvert$

Predefined correlation functions. Note: $d_j = s_j - x_j$ and $\zeta_j = \min\{1, \theta_j \lvert d_j \rvert\}$ for Spherical and Cubic correlation functions

$$\xi(\zeta_j) = \begin{cases} 1 - 15\zeta_j^2 + 30 * \zeta_j^3 & \text{for } 0 \leq \zeta_j \leq 0.2 \\ 1.25(1 - \zeta_j^{)3} & \text{for } 0.2 \leq \zeta_j \leq 1 \\ 0 & \text{for } \zeta_j \geq 1 \end{cases} \tag{10}$$

This class also support an user defined function.

```
def corr_model(x, s, params, dt, dx):
    ...
    if dt:
        return rx, drdt
    if dx:
        return rx, drdx
    return rx
```

where 'rx' is an array defining the correlation matrix between 'x' and 's'. 'drdt' and 'drdx' are derivative of correlation matrix w.r.t hyperparameter (θ) and sample space (x).

$$\mathrm{rx}_{ij} = \prod_{k=1}^{n} \mathbf{R}_k(x_{ik} - s_{jk})$$
$$\mathrm{drdt}_{ijk} = \frac{\partial \mathrm{rx}_{ij}}{\partial \theta_k}$$
$$\mathrm{drdx}_{ijk} = \frac{\partial \mathrm{rx}_{ij}}{\partial x_k}$$

- `corr_model_params`:
  A numpy array of size $1 \times n$ specifying the starting point of hyperparamters for Maximum Likelihood Estimator. Default value is an array of all ones.

- `op`:
  Indicator to solve MLE problem or not. If 'True', this class uses scipy.optimize.fmin_l_bfgs_b to solve optimization problem. It is a gradient-based optimization algorithm and uses `corr_model_params` as initial point for optimization problem. If 'False', `corr_model_params` will be directly use as hyperparamters. Default: 'True'.

- `n_opt`:
  An integer specifying the number of times to estimate maximum likelihood estimator with different random starting points. Default value is assigned as 1.

- `bounds`:
  An array or list of size $2 \times n$, specifying the bounds on hyperparameters.

These bounds are used to generate new random starting points, while estimating maximum likelihood solution. Random samples are generated using log-uniform distribution.

Krig Methods:

- `interpolate`:
  A function which takes samples and returns the value of surrogate model at the sample. If 'dy' is True, then this function returns value of surrogate model and mean square error at the sample.

```
    K = Krig(samples=S, values=Y, reg_model='Linear',
             corr_model='Gaussian')
    y, mse = K.interpolate(x, dy=True)
```

- `jacobian`:
  A function which takes samples and returns the gradient of surrogate model at the samples.

```
    K = Krig(samples=S, values=Y, reg_model='Linear',
             corr_model='Gaussian')
    y_grad = K.jacobian(x)
```

Examples:
Two examples illustrating the use of the `Krig` class are provided in the following Jupyter scripts.

- Krig_Example1.ipynb:
  In this example, the `STS` is used to generate 20 samples from a 1-D gamma probability distribution. The function values are evaluated using RunModel. Kriging class is used to create an approximate surrogate model using linear regression model and gaussian correlation model. Then plot is shown to compare the actual and surrogate model.

- Krig_Example2.ipynb:
  In this example, the `STS` is used to generate 196 samples from a 2-D uniform probability distribution. Kriging class is used to create an approximate surrogate model using quadratic regression model and exponential correlation model. Then 3-D plots show the comparison between the actual and surrogate model.

- Krig_Example3.ipynb:
  This example illustrate the use of user-defined regression and correlation model. `reg_model` and `corr_model` are functions instead of strings, which uses pre-defined models.

5.6 `StochasticProcess` Module (Coming in V2.0)

The `StochasticProcess` module consists of classes and functions to generate samples of Stochastic Processes from Power Spectrum, Bispectrums and Autocorrelation Functions. The generated Stochastic Processes can be transformed into other random variables. We can import the module into a Python script with the following command

```
from UQpy import StocahsticProcess
```

The `StochasticProcess` module has the following classes, each corresponding to a different method:

Class	**Method**
`SRM`	Spectral Representation Method
`BSRM`	Bispectral Representation Method
`KLE`	Karhunen Louve Expansion
`Translate`	Translate Gaussian into Non-Gaussian
`Inverse_Translate`	Translates Non-Gaussian into Gaussian

Each class can be imported individually into a python script. For example, the `SRM` class can be imported to a script using the following command:

```
from UQpy.StochasticProcess import SRM
```

The following subsections describe each class, their respective inputs and attributes, and their use.

5.6.1 `UQpy.StochasticProcess.SRM` (Coming in V2.0)

`SRM` is a class for generating Stochastic Processes by Spectral Representation Method from a prescribed Power Spectral Density Function. The `SRM` class is imported using the following command:

```
from UQpy.StochasticProcess import SRM
```

The attributes of the `SRM` class are listed below:

`SRM` Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
`nsamples`	Input	⋆	
`S`	Input	⋆	
`dw`	Input	⋆	
`nt`	Input	⋆	
`nw`	Input	⋆	
`case`	Input	⋆	
`g`	Input	⋆	
`samples`	Output		

Description of `SRM` Class Attributes:

Input Attributes:

- `nsamples`:
  A scalar integer value defining the the number of samples of the Stochastic Process to be generated.

- `S`:
  A numpy array defining the Power Spectral Density to be used for generation of the Stochastic Processes.

- `dw`:
  The length of the frequency discretisation to be used for the generation of the Stochastic Processes.

- `nt`:
  Specifies the number of time discretisations of the generated Stochastic Processes.

- `nw`:
  Specifies the number of frequency discretisations of the Power Spectrum.

- `case`:
  A String specifying if it is a univariate or multivariate Stochastic Process. Acceptable values are 'uni' for one variable case and 'multi' for multi variable case.

- `g`:
  A numpy array defining the Cross Power Spectral Density. It is only used in the 'multi' case.

Output Attributes:

- `samples`:
  A numpy array of samples following the Power Spectral Density.

Examples:
A bunch of example files illustrating the use of the `SRM` class are provided:

- SRM_1D_1V.ipynb:
  In this example, one-dimensional uni-variate Stochastic Processes are generated.

- SRM_1D_mV.ipynb:
  In this example, one-dimensional multi-variate Stochastic Processes are generated.

- SRM_nD_1V.ipynb:
  In this example, n-dimensional uni-variate Stochastic Processes are generated.

- SRM_nD_mV.ipynb:
  In this example, n-dimensional multi-variate Stochastic Processes are generated.

5.6.2 `UQpy.StochasticProcess.BSRM` (Coming in V2.0)

`BSRM` is a class for generating Stochastic Processes by BiSpectral Representation Method from a prescribed Power Spectral Density Function and a Bispectral Density Function. The `BSRM` class is imported using the following command:

```
from UQpy.StochasticProcess import BSRM
```

The attributes of the `BSRM` class are listed below:

BSRM Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
`nsamples`	Input	⋆	
`S`	Input	⋆	
`B`	Input	⋆	
`dt`	Input	⋆	
`dw`	Input	⋆	
`nt`	Input	⋆	
`nw`	Input	⋆	
`samples`	Output		

Description of `BSRM` Class Attributes:

Input Attributes:

- `nsamples`:
  A scalar integer value defining the the number of samples of the Stochastic Process to be generated.

- `S`:
  A numpy array defining the Power Spectral Density to be used for generation of the Stochastic Processes.

- `B`:
  A numpy array defining the BiSpectral Density to be used for generation of the Stochastic Processes.

- `dt`:
  The length of the time discretisation to be used for the generation of the Stochastic Processes.

- `dw`:
  The length of the frequency discretisation to be used for the generation of the Stochastic Processes.

- `nt`:
  Specifies the number of time discretisations of the generated Stochastic Processes.

- `nw`:
  Specifies the number of frequency discretisations of the Power Spectrum.

Output Attributes:

- `samples`:
  A numpy array of samples generated by the BiSpectral Representation Method.

Examples:
Example files illustrating the use of the `BSRM` class have been provided:

- BSRM_1D.ipynb:
  In this example, one-dimensional Stochastic Processes are generated by BSRM method.

- BSRM_nD.ipynb:
  In this example, n-dimensional Stochastic Processes are generated by BSRM method.

5.6.3 `UQpy.StochasticProcess.KLE` (Coming in V2.0)

`KLE` is a class for generating Stochastic Processes by Karhunen Louve Expansion from a prescribed Autocorrelation Function. The `BSRM` class is imported using the following command:

```
from UQpy.StochasticProcess import KLE
```

The attributes of the `KLE` class are listed below:

`KLE` Class Attribute Definitions

Attribute	**Input/Output**	**Required**	**Optional**
`nsamples`	Input	⋆	
`R`	Input	⋆	
`samples`	Output		

Description of `KLE` Class Attributes:

Input Attributes:

- `nsamples`:
  A scalar integer value defining the the number of samples of the Stochastic Process to be generated.

- `R`:
  A numpy array defining the Autocorrelation Function to be used for generation of the Stochastic Processes.

Output Attributes:

- `samples`:
  A numpy array of samples generated by the Karhunen Louve Expansion.

Examples:
An example files illustrating the use of the `KLE` class have been provided:

- KLE.ipynb:
  In this example, Stochastic Processes are generated by Karhunen Louve Expansion method.

5.6.4 `UQpy.StochasticProcess.Translation` (Coming in V2.0)

`Translate` is a class for translating Gaussian Stochastic Processes to Non-Gaussian Stochastic Processes. This class returns the non-Gaussian samples along with the distorted Aurocorrelated Function. The `Translate` class is imported using the following command:

```
from UQpy.StochasticProcess import Translate
```

The attributes of the `Translate` class are listed below:

`Translate` Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
`samples_g`	Input	⋆	
`R_g`	Input	⋆	
`marginal`	Input	⋆	
`params`	Input	⋆	
`samples_ng`	Output		
`R_ng`	Output		

Description of `Translate` Class Attributes:

Input Attributes:

- `samples_g`:
  Numpy array of Gaussian samples to be translated into specified non-Gaussian samples.

- `R_g`:
  Numpy array providing the Autocorrelation Function of the Gaussian Stochastic Processes.

- `marginal`:
  The name of the marginal distribution to which to be translated. It must follow the format discussed in the Distributions module.(Examples Jupyter script may be referred for further coherence)

- `params`:
  The parameters of the marginal distribution to which to be translated. It must follow the format discussed in the Distributions module.(Examples Jupyter script may be referred for further coherence)

Output Attributes:

- `samples_ng`:
  Numpy array of the translated Non-Gaussian samples.

- `R_ng`:
  Numpy array of the distorted Non-Gaussian Autocorrelation Function.

Examples:
An example files illustrating the use of the `Translate` class have been provided:

- Translate.ipynb:
  In this example, a Gaussian Stochastic Process has been translated into a Uniform[0, 1] process.

5.6.5 `UQpy.StochasticProcess.InverseTranslation` (Coming in V2.0)

`Inverse_Translate` is a class for translating Non-Gaussian Stochastic Processes back to Standard Gaussian Stochastic Processes. This class returns the non-Gaussian samples along with the distorted Aurocorrelated Function. The `Translate` class is imported using the following command:

```
from UQpy.StochasticProcess import InverseTranslation
```

The attributes of the `Translate` class are listed below:

`Translate` Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
`samples_ng`	Input	⋆	
`R_ng`	Input	⋆	
`marginal`	Input	⋆	
`params`	Input	⋆	
`samples`	Output		

Description of `BSRM` Class Attributes:

Input Attributes:

- `samples_g`:
  Numpy array of non-Gaussian samples to be translated into standard Gaussian samples.

- `R_ng`:
  Numpy array providing the Autocorrelation Function of the non-Gaussian Stochastic Processes.

- `marginal`:
  The name of the marginal distribution the Stochastic Process currently follows. It must follow the format discussed in the Distributions module.(Examples Jupyter script may be referred for further coherence)

- `params`:
  The parameters of the marginal distribution the Stochastic Process currently follows. It must follow the format discussed in the Distributions module.(Examples Jupyter script may be referred for further coherence)

Output Attributes:

- `samples_g`:
  Numpy array of the standard Gaussian samples.

- `R_ng`:
  Numpy array of the Gaussian Autocorrelation Function.

Examples:
An example files illustrating the use of the `Inverse_Translate` class have been provided:

- Inverse_Translate.ipynb:
  In this example, a non-Gaussian Stochastic Process is translated into a standard Gaussian Stochastic Process.

5.7 Transformations

Class	**Method**
`Correlate`	Induces correlation
`Decorrelate`	Removes correlation
`Nataf`	Nataf transformation
`InvNataf`	Inverse Nataf transformation

5.7.1 `UQpy.SampleMethods.Correlate`

`Correlate` is a class for inducing correlation in independent standard normal random variables. This is done using the standard Cholesy method as follows. Let $\mathbf{Y}$ denote an uncorrelated standard normal random vector and $\mathbf{Z}$ denote a standard normal random vector with positive definite correlation matrix $\mathbf{C_Z}$. Perform the Cholesky decomposition of $\mathbf{C_Z}$ such that:

$$\mathbf{C_Z} = \mathbf{U}\mathbf{U}^T \tag{11}$$

where $\mathbf{U}$ is a lower-triangular matrix.

Given the `nsamples`$\times$ `dimension` array, $\mathbf{y}$, of uncorrelated standard normal samples, the array $\mathbf{z}$ of samples possessing correlation $\mathbf{C_Z}$ is determined by:

$$\mathbf{z}^T = \mathbf{U}\mathbf{y}^T \tag{12}$$

The `Correlate` class is imported using the following command:

```
from UQpy.SampleMethods import Correlate
```

The attributes of the `Correlate` class are listed below:

`Correlate` Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
`input_samples`	Input	⋆	
`corr_norm`	Input	⋆	
`dimension`	Input	⋆	⋆
`samples_uncorr`	Output		
`samples`	Output		

A brief description of each attribute can be found in the table below:

`Correlate` Class Attributes			
Attribute*	**Type**	**Options**	**Default**
`input_samples`	*ndarray/object*	`SampleMethods` object or User-defined array	
`corr_norm`	*ndarray*	User-defined array	
`dimension`	*integer*	Inherited from `SampleMethods` object or User-defined scalar	
`samples_uncorr`	*ndarray*		
`samples`	*ndarray*		

* Note: If `input_samples` is a `SampleMethods` object, the `Correlate` object will inherit all attributes of that object.

Detailed Description of `Correlate` Class Attributes:

Input Attributes:

- `input_samples`:
  Contains the independent standard normal random samples on which to impose correlation.

  `input_samples` can be an object (instance of a `SampleMethods` class) or an array.

  If `input_samples` is an instance of a `SampleMethods` class, then the `Correlate` class inherits all of its attributes and the correlation is induced on the samples contained in the attribute `input_samples.samples`.

  If `input_samples` is a `numpy` array, then the correlation is induced directly on `input_samples`. The number of samples is given by `nsamples=input_samples.shape[0]`.

- `corr_norm`:
  A `numpy` array containing the correlation matrix $\mathbf{C}$ for the random variables.

`corr_norm` must be a symmetric positive definite array of size `dimension` × `dimension` and satisfy:

`corr_norm[i, j] = 1` for `i = j`.

`0 < corr_norm[i, j] < 1` for `i` ≠ `j`.

`corr_norm[i,j] = corr_norm[j,i]`

- `dimension`:
  A scalar integer value defining the dimension of the random variables.

  If `input_samples` is a `SampleMethods` object then `dimension` is not required since `input_samples` already has the attribute `input_samples.dimension`.

  If `input_samples` is a `numpy` array, `dimension` must be specified.

Output Attributes:

- `samples_uncorr`:
  A numpy array of dimension `nsamples` × `dimension` containing the original uncorrelated standard normal samples.

  If `input_samples` is an array then `samples_uncorr=input_samples`.

  if `input_samples` is a `SampleMethods` object, then `samples_uncorr=input_samples.samples`.

- `samples`:
  A `numpy` array of dimension `nsamples` × `dimension` containing the correlated standard normal samples with correlation defined in `corr_norm`.

Examples:
An example illustrating the use of the `Correlate` class is provided in the following Jupyter script.

- Correlate.ipynb:
  In this example, 1000 2-dimensional standard normal samples are correlated according to a specified correlation matrix. The input samples are specified using both the `MCS` class and as a `numpy` array generated using `scipy.stats`.

5.7.2 UQpy.SampleMethods.Decorrelate

`Decorrelate` is a class for removing correlation from a `nsamples`$\times$`dimension` array, $\mathbf{z}$, of standard normal random samples with correlation matrix $\mathbf{C_Z}$. This is performed by simply inverting the expression in Eq. (12) as:

$$\mathbf{y}^T = \mathbf{U}^{-1}\mathbf{z}^T \tag{13}$$

to obtain the `nsamples`$\times$`dimension` array, $\mathbf{y}$, of uncorrelated standard normal samples.

The `Decorrelate` class is imported using the following command:

```
from UQpy.SampleMethods import Decorrelate
```

The attributes of the `Decorrelate` class are listed below:

`Decorrelate` Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
`input_samples`	Input	⋆	
`corr_norm`	Input	⋆	
`dimension`	Input	⋆	⋆
`samples_corr`	Output		
`samples`	Output		

A brief description of each attribute can be found in the table below:

`Decorrelate` Class Attributes			
Attribute*	**Type**	**Options**	**Default**
`input_samples`	*ndarray/object*	Object of class `Correlate` or User-defined array	
`corr_norm`	*ndarray*	Inherited from `Correlate` object or User-defined array	
`dimension`	*integer*	Inherited from `Correlate` object or User-defined scalar	
`samples_corr`	*ndarray*		
`samples`	*ndarray*		

* Note: If `input_samples` is a `Correlate` object, the `Decorrelate` object will inherit all attributes of that object.

Detailed Description of `Decorrelate` Class Attributes:

Input Attributes:

- `input_samples`:
  Contains the correlated standard normal samples whose correlation will be removed.

  `input_samples` can be an object (instance of the `Correlate` class) or a `numpy` array.

  If `input_samples` is an instance of `Correlate`, then the `Decorrelate` class inherits all of its attributes and the decorrelation is performed on the attribute `input_samples.samples`.

  If `input_samples` is a `numpy` array, then the decorrelation is performed directly on `input_samples`. The number of samples is given by `nsamples=input_samples.shape[0]`.

- `corr_norm`:
  A `numpy` array containing the correlation matrix **C** for the random variables.

  If `input_samples` is an object of the `Correlate` class, then `corr_norm` is inherited this class.

  If `input_samples` is a `numpy` array, then `corr_norm` must be specified.

  `corr_norm` must be a symmetric positive definite array of size `dimension` × `dimension` and satisfy:

  `corr_norm[i, j] = 1` for `i = j`.

  `0 < corr_norm[i, j] < 1` for `i` ≠ `j`.

  `corr_norm[i,j] = corr_norm[j,i]`

- `dimension`:
  A scalar integer value defining the dimension of the random variables.

If `input_samples` is a `Correlate` object then `dimension` may not be required since `input_samples` may already have the attribute `input_samples.dimension`.

If `input_samples` is a `numpy` array, `dimension` must be specified.

Output Attributes:

- `samples_corr`:
  A `numpy` array of dimension `nsamples` $\times$ `dimension` containing the original correlated samples.

  If `input_samples` is an array then `samples_corr=input_samples` and if `input_samples` is an object of the `Correlate` class then `samples_corr=input_samples.samples`.

- `samples`:
  A `numpy` array of dimension `nsamples` $\times$ `dimension` containing the uncorrelated standard normal samples.

Examples:
An example illustrating the use of the `Decorrelate` class is provided in the following Jupyter script.

- Decorrelate.ipynb:
  In this example, 1000 2-dimensional correlated standard normal samples are generated using the `Correlate` class and using the `scipy.stats` package. The samples from each are decorrelate using the `Decorrelate` class.

5.7.3 `UQpy.SampleMethods.InvNataf`

`InvNataf` is a class for transforming standard normal random samples to a prescribed non-Gaussian distribution using the inverse Nataf transformation.

Theory
Let $\mathbf{Z}$ denote an n-dimensional standard normal random vector and let $F_i(x_i), i = 1, \ldots, n$ be the marginal cumulative distribution functions of the n correlated non-Gaussian random variables X_i. According to the Nataf transformation, the non-Gaussian random vector, $\mathbf{X}$, following $F_i(x_i)$ is defined component-wise through the transformation:

$$x_i = F_i^{-1}(\Phi(z_i)) \tag{14}$$

where $\Phi(x)$ is the standard normal cumulative distribution function.

When the random vector $\mathbf{Z}$ has correlated components possessing correlation matrix $\mathbf{C_Z}$ and correlation coefficients ρ_{ij} between components Z_i and Z_j, the transformation in Eq. (14) causes a so-called *correlation distortion* such that the correlation coefficient between the non-Gaussian variables X_i and X_j, denoted ξ_{ij}, is not equal to the correlation between the Gaussian variables ($\rho_{ij} \neq \xi_{ij}$). The non-Gaussian correlation coefficient, ξ_{ij}, can be determined from the Gaussian correlation coefficient, ρ_{ij}, through the following integral:

$$\xi_{ij} = \frac{1}{\sigma_{X_i}\sigma_{X_j}} \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \left(F_i^{-1}(\Phi(z_i)) - \mu_{X_i}\right)\left(F_j^{-1}(\Phi(z_j)) - \mu_{X_j}\right) \phi_2(z_i, z_j; \rho_{ij}) dz_i dz_j \quad (15)$$

where $\phi_2(\cdot)$ is the joint Gaussian pdf.

When conducting probabilistic modeling using the inverse Nataf transformation (particularly when performing the first and second order reliability method FORM/SORM, see Section **??**), it is useful to know the Jacobian of the transformation in Eq. (14). Let us rewrite Eq. (14) as:

$$F_i(x_i) = \Phi(z_i) \quad (16)$$

Taking the derivative of Eq. (16) yields:

$$\begin{aligned} \frac{\partial F_i}{\partial x_i} =& \frac{\partial}{\partial x_i}\left(\Phi(z_i)\right) \\ f_i(x_i) =& \frac{\partial \Phi(z_i)}{\partial x_i}\frac{\partial z_i}{\partial x_i} \\ f_i(x_i) =& \phi(z_i)\frac{\partial z_i}{\partial x_i} \end{aligned}$$

Rearranging this equation, we arrive at the Jacobian of the inverse Nataf transformation with components

$$J_{x_i,z_i} = \frac{\partial x_i}{\partial z_i} = \frac{\phi(z_i)}{f_i(x_i)} \quad (17)$$

The Jacobian of the inverse Nataf transformation is assembled as a diagonal matrix given by:

$$\mathbf{J_{xz}} = \frac{\partial \mathbf{x}}{\partial \mathbf{z}} = \left[\frac{\phi(z_i)}{f_i(x_i)}\right] \quad (18)$$

It is more common, in practice, to combine the steps of correlating the variables and mapping them to the non-Gaussian distribution through the inverse Nataf. In other words, letting $\mathbf{y}$ denote an n-dimensional vector of uncorrelated standard normal random variables, we can express the Jacobian of the transformation from $\mathbf{y}$ to $\mathbf{x}$ by:

$$\mathbf{J_{xy}} = \frac{\partial \mathbf{x}}{\partial \mathbf{y}} = \frac{\partial \mathbf{x}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{y}} \tag{19}$$

where, by applying Eqs. (12) and (18), we see that:

$$\mathbf{J_{xy}} = \frac{\partial \mathbf{x}}{\partial \mathbf{y}} = \mathbf{U} \left[\frac{\phi(z_i)}{f_i(x_i)} \right] \tag{20}$$

where $\mathbf{U}$ is the lower triangular matrix resulting from the Cholesky decomposition of $\mathbf{C_Z}$ in Eq. (11).

The Jacobian in Eq. (20), which combines the correlation and inverse Nataf steps, is the one computed by the `InvNataf` class.

Using the `InvNataf` Class

The `InvNataf` class is imported using the following command:

```
from UQpy.SampleMethods import InvNataf
```

The attributes of the `InvNataf` class are listed below:

`InvNataf` Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
`input_samples`	Input		⋆
`corr_norm`	Input	⋆	
`dist_name`	Input	⋆	
`dist_params`	Input	⋆	
`dimension`	Input	⋆	⋆
`samplesN01`	Output		
`samples`	Output		
`corr`	Output		
`jacobian`	Output		

A brief description of each attribute can be found in the table below:

`InvNataf` Class Attributes			
Attribute	**Type**	**Options**	**Default**
`input_samples`	*ndarray/object*	`SampleMethods` object or User-defined array	`None`
`corr_norm`	*ndarray*	Inherited from `SampleMethods` object or User-defined array	Identity Matrix $\mathbf{I}_{\texttt{dimension}}$
`dimension`	*integer*	Inherited from `SampleMethods` object or User-defined integer	
`dist_name`	*function/string list*	`name` attribute from `Distributions` class See Section 6.1	
`dist_params`	*ndarray list*	See Section 6.1	
`samplesN01`	*ndarray*		
`samples`	*ndarray*		
`corr`	*ndarray*		
`jacobian`	*ndarray list*		

Detailed Description of `InvNataf` Class Attributes:

Input Attributes:

- `input_samples`:
  Contains the samples to be transformed. The samples need to be standard normal samples i.e $\sim N(0,1)$.

  `input_samples` can be a `SampleMethods` object or a `nsamples`× `dimension` `numpy` array. The inverse Nataf transformation is applied to the `samplesN01` object. Depending on the type of `input_samples`, `samplesN01` is assigned as follows:

  - If `input_samples` is a `SampleMethods` object, then the `InvNataf` class inherits all the attributes of that object and `samplesN01` = `input_samples.samples`

  - If `input_samples` is an array, then `samplesN01` = `input_samples`.

  If `input_samples` is not provided, then `InvNataf` calculates the correlation distortion of the standard normal correlation matrix `corr_norm` from Eq. (15).

  The default value of `input_samples` is `None`.

- `dimension`:
  A scalar integer value defining the dimension of the random variables.

  If `input_samples` is a `SampleMethods` object, then `dimension` may not be required since `input_samples` may already have the attribute `input_samples.dimension`.

  If `input_samples` is a `numpy` array, `dimension` must be specified.

- `corr_norm`:
  A `numpy` array containing the correlation matrix $\mathbf{C}$ for the standard normal random variables.

  `corr_norm` must be a symmetric positive definite array of size `dimension` × `dimension` and satisfy:

  `corr_norm[i, j] = 1` for `i = j`.

  `0 < corr_norm[i, j] < 1` for `i` ≠ `j`.

  `corr_norm[i,j] = corr_norm[j,i]`

  If `input_samples` is an object of type `Correlate` then `corr_norm` is inherited from this object.

  The default value of `corr_norm` is the `dimension`×`dimension` identity matrix $\mathbf{I}_{\texttt{dimension}}$.

- `dist_name`:
  Specifies the name of the marginal distribution that each transformed random variable.

  `dist_name` may be a string or a list of strings of length `dimension`.

  For each dimension `i`, `dist_name[i]` must be a string specifying a distribution defined in the `Distributions` module (see Sec. 6.1). To use a custom distribution, set `dist_name[i]` = 'custom_dist' to use the custom distribution assignment option in the `Distributions` module (again, see Sec. 6.1).

If `dist_name` is a string (or a list of length one) and `dimension` > 1, then `dist_name` is converted into a list of length `dimension` with each component having identical distribution name.

`dist_name` must be specified. There is no default value.

- `dist_params`:
  Specifies the parameters for each marginal distribution in `dist_name` as defined in the `Distributions` module (see Sec. 6.1).

  Each set of parameters is defined as a `numpy` array. `dist_params` is a list of arrays, with each item in the list corresponding to the associated random variable.

  If `dist_params` is an array (or a list of length one), then `dist_params` is converted to a list of length `dimension` with each component having the same parameters.

  `dist_params` must be specified. There is no default value.

Output Attributes:

- `samplesN01`:
  A numpy array of dimension `nsamples` $\times$ `dimension` containing the correlated or uncorrelated standard normal samples that have have been transformed.

  If `input_samples` = `None`, `samplesN01` is not returned.

  If `input_samples` is a `SampleMethods` object, then `samplesN01` = `SampleMethods.samples`. If `input_samples` is an array then `samplesN01` = `input_samples`.

- `samples`:
  A `numpy` array of dimension `nsamples` $\times$ `dimension` containing the correlated or uncorrelated transformed samples follwing the prescribed distribution.

If `input_samples` = `None`, `samples` is not returned.

- `corr`:
  A `numpy` array containing the transformed/distorted correlation matrix.

  If `corr_norm` = `None` or `corr_norm` = $\mathbf{I}$, where $\mathbf{I}$ is the identity matrix, then `corr` = `corr_norm` = $\mathbf{I}$.

- `jacobian`:
  A list of `numpy` arrays containing the Jacobian of the transformation evaluated at each sample.

Examples:
Three examples illustrating the use of the `Nataf` class are provided in the following Jupyter scripts.

- InvNataf - Example 1.ipynb:
  In this example, the `InvNataf` class is used in order to transform 1000 samples of 2 uncorrelated standard normal variables to a lognormal and a gamma distribution. The example illustrates the transformation for samples drawn using the `MCS` class and for samples specified as a `numpy` array.

- InvNataf - Example 2.ipynb:
  In this example, the `InvNataf` class is used in order to transform 1000 samples of 2 correlated standard normal variables to a lognormal and a gamma distribution. The example illustrates the transformation for samples drawn using the `MCS` class and correlated using the `Correlate` class and for samples specified as a `numpy` array.

- InvNataf - Example 3.ipynb:
  In this example, the `InvNataf` class is used to calculate the correlation distortion for the transformation of two correlated random variables from a standard normal to a lognormal distribution.

5.7.4 `UQpy.SampleMethods.Nataf`

`Nataf` is a class for transforming non-Gaussian random variables to equivalent standard normal space. The `Nataf` class is imported using the following command:

```
from UQpy.SampleMethods import Nataf
```

The attributes of the Nataf class are listed below:

Nataf Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
input_samples	Input	⋆	⋆
dimension	Input	⋆	⋆
corr	Input	⋆	
dist_name	Input	⋆	⋆
dist_params	Input	⋆	⋆
samplesNG	Output		
samples	Output		
corr_norm	Output		
jacobian	Output		

A brief description of each attribute can be found in the table below:

Nataf Class Attributes			
Attribute	**Type**	**Options**	**Default**
input_samples	*ndarray/object*	Attribute of class MCS, LHS, STS, Correlate, Nataf or User-defined array	None
corr	*ndarray*	Attribute of class Nataf or User-defined array	
dimension	*integer*	Attribute of class MCS, LHS, STS, Correlate, Nataf or User-defined scalar	
dist_name	*function/string list*	See Distributions Module or User-defined function	
dist_params	*ndarray list*		
samplesNG	*ndarray*		
samples	*ndarray*		
corr_norm	*ndarray*		
jacobian	*ndarray list*		

Detailed Description of Nataf Class Attributes:

Input Attributes:

- input_samples:
  Contains the samples to be transformed to standard normal samples.

`input_samples` can be an object of type `MCS, LHS, STS, Correlate, InvNataf` or a numpy array.

If `input_samples` is an object of type `MCS, LHS, STS, Correlate, Nataf`, then the `InvNataf` class inherits all the attributes of the class and the transformation is performed to the attribute `.samples` of the class.

If `input_samples` is an array then the transformation is performed directly to the `input_samples`. The number of samples is given by `nsamples=input_samples.shape[0]`.

If `input_samples` is not provided then class `Nataf` calculates the correlation matrix `corr_norm` in the standard normal space.

The default value of `input_samples` is `None`.

- `dimension`:
  A scalar integer value defining the dimension of the random variables.
- `corr`:
  A numpy array showing the correlation coefficients between the non-Gaussian random variables.

  `corr` must be an array of size `dimension` × `dimension` and satisfy:

  `corr[i, j] = 1` for `i = j`.

  `corr[i, j] < 1` for `i` ≠ `j`.

  if `input_samples` is an object of type `Nataf` then `corr` is an attribute of this class.

  if `input_samples` is an object of type `MCS, LHS, STS` then `corr` is set to be the identity matrix `I_dimension`.

- `dist_name`:
  Defines the name of the marginal distribution that each standard normal random variable will be transformed to.

  `dist_name` may be a string, a function, or a list of strings/functions.

  If `dist_name[i]` is a string, the distribution is matched with one of the available functions in the `Distributions` module (see Sec. 6.1) or the 'custom_dist.py' (again see Sec. 6.1).

  if `dist_name[i]` is a function, it must be defined in the user's Python script and passed directly as a function.

  `dist_name` can contain an arbitrary combination of strings and functions.

  If `dist_name` is a string or function (or a list of length one) and `dimension` > 1, then `dist_name` is converted into a list of length `dimension` with each variable having the distribution.

  if `data` is not an object of type `MCS, LHS, STS, InvNataf` then `dist_name` must be specified. There is no default value.

- `dist_params`:
  Specifies the parameters for each marginal distribution in `dist_name`.

  Each set of parameters is defined as a numpy array. `dist_params` is a list of arrays, with each item in the list corresponding to the associated random variable.

  If `dist_params` is an array (or a list of length one), then `dist_params` is converted to a list of length `dimension` with each variable having the same parameters.

  if `input_samples` is not an object of type `MCS, LHS, STS, InvNataf` then `dist_params` must be specified. There is no default value.

Output Attributes:

- `samplesNG`:
  A numpy array of dimension `nsamples` $\times$ `dimension` containing the correlated or uncorrelated non-Gaussian samples. It is an output of the class only if `data` is not `None`.

  If `input_samples` is an object of type `MCS, LHS, STS, Correlate, InvNataf` then `samplesNG .samples`. If `input_samples` is an array then `samplesNG=input_samples`.

- `samples`:
  A numpy array of dimension `nsamples` $\times$ `dimension` containing the correlated or uncorrelated standard normal samples. It is an output of the class only if `input_samples` is not `None`.

- `corr_norm`:
  A numpy array containing the correlation matrix in the standard normal space.

  if `data` is an object of type `MCS, LHS, STS, Correlate` then `corr` = `corr_norm` = `I_dimension`.

- `jacobian`:
  A list containing the jacobian of the transformation for each sample as an numpy array.

Examples:
An example illustrating the use of the `Correlate` class is provided in the following Jupyter script.

- Nataf - Example 1.ipynb:
  In this example, `Nataf` class is used in order to transform 2 correlated lognormal variables to two standard normal random variables.
- Nataf - Example 2.ipynb:
  In this example, `Nataf` class is used to perform the Iterative Translation Approximation Method (ITAM) [11] to estimate the underlying Gaussian correlation from known values of the correlation for lognormal random variables.

6 Support Modules

The modules detailed in Section 4 form the core of `UQpy` and its primary capabilities. In support of these primary modules are two additional modules that provide capabilities that are generally used throughout the primary modules. These two support modules are described herein.

6.1 `Distributions` Module

The `Distributions` module is the structure through which probability distributions and their related operations are defined in `UQpy`. This includes functions for computing probability densities, cumulative distributions and their inverses, moments, the logarithms of the probability densities as well as parameter estimates for generic data for common distribution types.

The `Distributions` module is imported in a Python script using the following command:

```python
from UQpy import Distributions
```

The `Distributions` module contains three classes: The `Distribution` class, the `SubDistribution` class, and the `Copula` class. The `Distribution` class is the parent class of the module, which calls the `SubDistribution` and `Copula` classes as necessary to construct a `Distribution` object.

Distributions in `UQpy` can generally be categorized in one of three types: 1. Marginal distributions for a single random variable; 2. Joint distributions with independent random variables; 3. Joint distributions with dependent random variables and. The user can define a probability distribution object by providing a name (see supported distributions in `SubDistribution` class or custom distribution) and a dependency structure through the `Copula` class (optional). This class possesses the following attributes:

`Distribution` Class Attribute Definitions			
Attribute	Input/Output	**Type**	**Required**
`dist_name`	Input	*string/list*	*
`copula`	Input	*string*	

The `SubDistribution` class, has the following attribute:

`SubDistribution` Class Attribute Definitions			
Attribute	Input/Output	**Type**	**Required**
`dist_name`	Input	*string*	*

and the following methods:

`SubDistribution` Class Methods	
Method	**Type**
`pdf`	*function*
`rvs`	*function*
`cdf`	*function*
`icdf`	*function*
`log_pdf`	*function*
`fit`	*function*
`moments`	*function*

`Copulas` class having the following attributes:

`Copulas` Class Attribute Definitions			
Attribute	Input/Output	**Type**	**Required**
`copula_name`	Input	*string*	*
`dist_name`	Input	*list*	*

and the following methods:

`Copula` Class Methods	
Method	**Type**
`pdf`	*function*

With the exception of the custom distribution, the `SubDistribution` class simply repackages certain distributions from the `scipy.stats` package in a way that is convenient to use within `UQpy`. A brief description of each attribute of the `Distribution` class is presented next.

Detailed Description of `Distribution` Class Attributes:

Input Attributes:

- `dist_name`:
  A *string* or a *list* of *strings* designating the distribution name (available distributions are shown in the table below) and the distribution type (univariate/multivariate).
  - If `dist_name` is a *string* → univariate distribution.
  - If `dist_name` is a *list* → multivariate distribution.

`dist_name` must be specified. `Distribution` does not have a default distribution type.

- `copula`:
  Defines the dependency between dimensions and in order to use it the `dist_name` should be given as a *list*. The available copulas are shown in the table below.

Supported `Copulas` in `UQpy`	
Name	**Parameters**
"Gumbel"	$\theta \in [1, +\infty)$

  `copula` is optional. The default copula value is None.

Distribution Methods
The instantiating of a `Distribution` object can be made with the following ways:

```
Distribution(name=dist_name)

Distribution(name=[dist_name_1, dist_name_2, ...])

Distribution(name=[dist_name_1, dist_name_2, ...],
copula=copula)
```

The `Distribution` object gives access to the following functions: `pdf, cdf, icdf, rvs, moments, log_pdf, fit`.

- `pdf`:
  A function that returns the probability density function at a specified value or values x. Note that the parameters of the distribution must be passed into the `pdf` function.

  If the distribution is univariate (or the special case of multivariate normal) the function is called as follows:

  ```
  Distribution(dist_name).pdf(x,params)
  ```

  If the distribution is multivariate the function is called as follows:

  ```
  Distribution([dist_name_1,...]).pdf(x, [params_1,...])
  ```

Note that `[params_1, params_2, ...]` correspond to distribution models `[dist_name_1, dist_name_2,...]`. In this case, the output of the `pdf` function is the product of the marginal `pdf`s

$$\prod_i \texttt{Distribution(dist\_name\_i).pdf(x[:, i], params\_i)}$$

where `params` in both cases is given as a *list*.

- `rvs`:
  A function that draws random samples from the specified distribution. Note that the parameters of the distribution must be passed into the `rvs` function and the number of samples (`nsamples`) must be specified.

  For a univariate distribution the function is called as follows:

  `Distribution(dist_name).rvs(params, nsamples)`

  If the distribution is multivariate the function is called as follows:

  `Distribution([dist_name_1,...]).rvs([params_1,...], nsamples)`

  In this case the output vector is defined as

  $$\mathrm{x}[:, \mathrm{i}] = \texttt{Distribution(dist\_name\_i).rvs(params\_i, nsamples)}$$

- `cdf`:
  A function that returns the cumulative distribution function at a specified value x. Note that the parameters of the distribution must be passed into the `cdf` function.

  For a univariate distribution the function is called as follows:

  `Distribution(dist_name).cdf(x,params)`

  If the distribution is multivariate the function is called as follows:

  `Distribution([dist_name_1,...]).cdf(x, [params_1,...])`

  In this case the output is a *list* with entries the values of `cdf` calculated at `x` for every distribution model defined in `[dist_name_1,dist_name_2,...]`.

- `icdf`:
  A function that returns the inverse cumulative distribution function at a specified value or values $x \in [0, 1]$. Note that the parameters of the distribution must be passed into the `icdf` function.

For a univariate distribution the function is called as follows:

`Distribution(dist_name).icdf(x,params)`

If the distribution is multivariate the function is called as follows:

`Distribution([dist_name_1,...]).icdf(x, [params_1,...])`

In this case the output is a *list* with entries the values of `icdf` calculated at `x` for every distribution model defined in `[dist_name_1,dist_name_2,...]`.

- `log_pdf`:
  A function that returns the logarithm of the probability density function at a specified value or values x. Note that the parameters of the distribution must be passed into the `log_pdf` function.

  If the distribution is univariate the function is called as follows:

  `Distribution(dist_name).log_pdf(x,params)`

  If the distribution is multivariate the function is called as follows:

  `Distribution([dist_name_1,...]).log_pdf(x, [params_1,...])`

  In this case, the output of the `log_pdf` function is the sum of the marginal `log_pdf`s

  $$\sum_i \texttt{Distribution(dist\_name\_i).log\_pdf(x[:, i], params\_i)}$$

- `fit`:
  A function that fits the parameters of the specified distribution to user-specified data y. Note that the parameters of the distribution that are returned follow the conventions of `scipy.stats`, which for some distributions may be inconsistent with the parameters specified in `UQpy`.

  For a univariate distribution the function is called as follows:

  `Distribution(dist_name).fit(x,params)`

  If the distribution is multivariate the function is called as follows:

  `Distribution([dist_name_1,...]).fit(x, [params_1,...])`

  In this case the output is a *list* with entries the values of `fit` calculated at `x` for every distribution model defined in `[dist_name_1, dist_name_2,...]`.

- `moments`:
  A function that returns the mean, variance, skewness, and kurtosis, of a specified distribution. Note that the parameters of the distribution must be

passed into the `moments` function.

For a univariate distribution the function is called as follows:

```
Distribution(dist_name).moments(params)
```

If the distribution is multivariate the function is called as follows:

```
Distribution([dist_name_1,...]).moments([params_1,...])
```

In this case the output is a *list* with entries the values of `moments` calculated at `x` for every distribution model defined in `[dist_name_1,dist_name_2,...]`.

Available Distributions in `UQpy`		
Distribution	**Name**	**Parameters**
`Beta`	"beta"	$[a, b]$ $a, b > 0, (a < b) \in \mathbb{R}$ Fixed: loc = 0, scale = 1
`Binomial`	"binomial"	$[n, p]$ $n \in \mathbb{N}_0, p \in [0, 1]$
`Cauchy`	"cauchy"	$[loc, scale]$ $loc, scale > 0$
`Chi-Squared`	"chisquare"	$[df, loc, scale]$
`Exponential`	"exponential"	$[loc, scale]$
`Gamma`	"gamma"	$[a, loc, scale]$ $a > 0$
`Generalized Extreme Value`	"genextreme"	$[c, loc, scale]$
`Inverse Gaussian`	"inv_gauss"	$[\mu, loc, scale]$
`Laplace`	"laplace"	$[loc, scale]$ $scale > 0$
`Levy`	"levy"	$[loc, scale]$ $scale > 0$
`Logistic`	"logistic"	$[loc, scale]$ $scale > 0$
`Lognormal`	"lognormal"	$[\sigma, loc, \mu]$ $s = \sigma$, $loc = loc$, $scale = \mu$, $\sigma > 0$
`Maxwell-Boltzmann`	"maxwell"	$[loc, scale]$ $scale > 0$
`Multivariate Normal`	"mvnormal"	$[\mathbf{M}, \mathbf{C}]$ $mean = \mathbf{M}, cov = \mathbf{C}$
`Normal(Gaussian)`	"normal" or "gaussian"	$[\mu, \sigma]$ $loc = \mu$, $scale = \sigma$ $\sigma > 0$
`Pareto`	"pareto"	$[b, loc, scale]$ $b, scale > 0$
`Rayleigh`	"rayleigh"	$[loc, scale]$ $scale > 0$
`Truncated Normal`	"truncnorm"	$[a, b, loc, scale]$ $a = \left(\frac{clip\_low - \mu}{\sigma}\right)$, $b = \left(\frac{clip\_high - \mu}{\sigma}\right)$ $loc = \mu$, $scale = \sigma$
`Uniform`	"uniform"	$[a, b]$ $loc = a$, $scale = b - a$ $b > a$

User-defined Distributions:

Other distributions can be easily added by defining the appropriate functions in a python script (`.py`). These functions must be consistent with those listed in the "`Distribution` Class Methods" table above.

Description of a (.py) script for a custom distribution
The user may define custom functions that compute the pdf, cdf, inverse cdf, or log_pdf at a specified value for the distribution as well as functions to generate samples, fits the distribution parameters, and returns the moments of the distribution. These functions should be defined within a single python script (`.py`). For compatibility with `UQpy`, the name of each function, must be specified as **`pdf`**, **`cdf`**, **`icdf`**, **`log_pdf`**, **`fit`** or **`moments`** in accordance with the conventions of the `Distribution` class. Each function is required to take inputs as prescribed above in the list of *Output Attributes* for the `Distribution` class.

Examples:
An example illustrating the use of the `Distribution` class with a built-in distribution is provided in the following Jupyter script.

- Distributions.ipynb:
  In this example, we explore the use of the `Distribution` class with a lognormal distribution.

An example illustrating the use of the `Distribution` class with a custom distribution provided through `custom_dist.py` is provided in the following Jupyter script.

- Custom_Distribution.ipynb:
  In this example, we explore the use of the `Distribution` class with a custom Weibull distribution.

6.2 `Utilities` Module

The `Utilities` module contains functionality for all the supporting methods in `UQpy`. It is imported in a python script using the following command:

```
from UQpy import Utilities
```

The `Utilities` module consists of various `functions`, each used for different purposes and can be called as:

```
from UQpy.Utilities import function
```

A list of the available functions that can be found in `Utilities` with a short description and the class in which is used is presented next.

List of available functions in module `Utiliti`	
Name	**Description**
`transform_ng_to_g`	Transform non-Gaussian to G
`transform_g_to_ng`	Transform Gaussian to non-G
`itam`	Iterative Translation Approxi
`run_corr`	Correlates standard normal v
`run_decorr`	Decorrelates standard normal
`correlation_distortion`	Evaluate the modified correla
`bi_variate_normal_pdf`	Evaluate the values of the bi-
`_get_a_plus`	A supporting function for the
`_get_ps`	A supporting function for the
`_get_pu`	A supporting function for the
`nearest_psd`	Compute the nearest positive
`nearest_pd`	Find the nearest positive-defi
`estimate_psd`	Estimate the Power Spectrum
`s_to_r`	Transform the power spectrum
`r_to_s`	Transform the autocorrelation
`is_pd`	Returns true when input is p
`resample` Perform some diagnostics on outputs of MCMC and IS height	Resample a set of samples ac

7 Adding new classes to `UQpy`

Adding new capabilities to `UQpy` is as simple as adding a new class to the appropriate module and importing the necessary packages into the module. Further details will be provided in the future as `UQpy` coding practices are formally established.

References

[1] J. M. Satagopan A. E. Raftery, M. A. Newton and P. N. Krivitsky. Estimating the integrated likelihood via posterior simulation using the harmonic mean identity. In *Bayesian Statistics 8*, pages 1–45, 2007.

[2] Siu-Kui Au and James L. Beck. Estimation of small failure probabilities in high dimensions by subset simulation. *Probabilistic Engineering Mechanics*, 16(4):263–277, oct 2001.

[3] K. P. Burnham and D. R. Anderson. Springer-Verlag, 2002.

[4] Robert Cimrman. SfePy - write your own FE application. In Pierre de Buyl and Nelle Varoquaux, editors, *Proceedings of the 6th European Conference on Python in Science (EuroSciPy 2013)*, pages 65–70, 2014. http://arxiv.org/abs/1404.6391.

[5] W.N. Edeling, R.P. Dwight, and P. Cinnella. Simplex-stochastic collocation method with improved scalability. *Journal of Computational Physics*, 310:301 – 328, 2016.

[6] A. Gelman, J.B. Carlin, Stern H.S., and M.D. Rubin. 2004.

[7] Jonathan Goodman and Jonathan Weare. Ensemble samplers with affine invariance. *Communications in applied mathematics and computational science*, 5(1):65–80, 2010.

[8] M. Grigoriu. Reduced order models for random functions. Application to stochastic problems. *Applied Mathematical Modelling*, 33(1):161–175, 2009.

[9] W K Hastings. Monte Carlo Sampling Methods Using Markov Chains and Their Applications. *Biometrika*, 57(1):97–109, 1970.

[10] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, 21(6):1087, 1953.

[11] M.D. Shields, G. Deodatis, and P. Bocchini. A simple and efficient methodology to approximate a general non-gaussian stationary stochastic process by a translation process. *Probabilistic Engineering Mechanics*, 26(4):511 – 519, 2011.

[12] Michael D. Shields. Adaptive monte carlo analysis for strongly nonlinear stochastic systems. *Reliability Engineering System Safety*, 175:207 – 224, 2018.

[13] Michael D. Shields, Kirubel Teferra, Adam Hapij, and Raymond P. Daddazio. Refined stratified sampling for efficient monte carlo based uncertainty quantification. *Reliability Engineering System Safety*, 142:310 – 325, 2015.

[14] O. Tange. Gnu parallel 2018. 2018.