UQpy - Uncertainty Quantification with Python

Michael D. Shields[*], Dimitris G. Giovanis[†]
Aakash Bangalore-Satish, Mohit Chauhan, Lohit Vandanapu,
Jiaxin Zhang

Shields Uncertainty Research Group (SURG)

Johns Hopkins University, USA

Version 1.1.0

[*] 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 6) 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.

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

Module	Class	Description	Version
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
	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
Reliability	SubsetSimulation	Subset Simulation	1.0.0

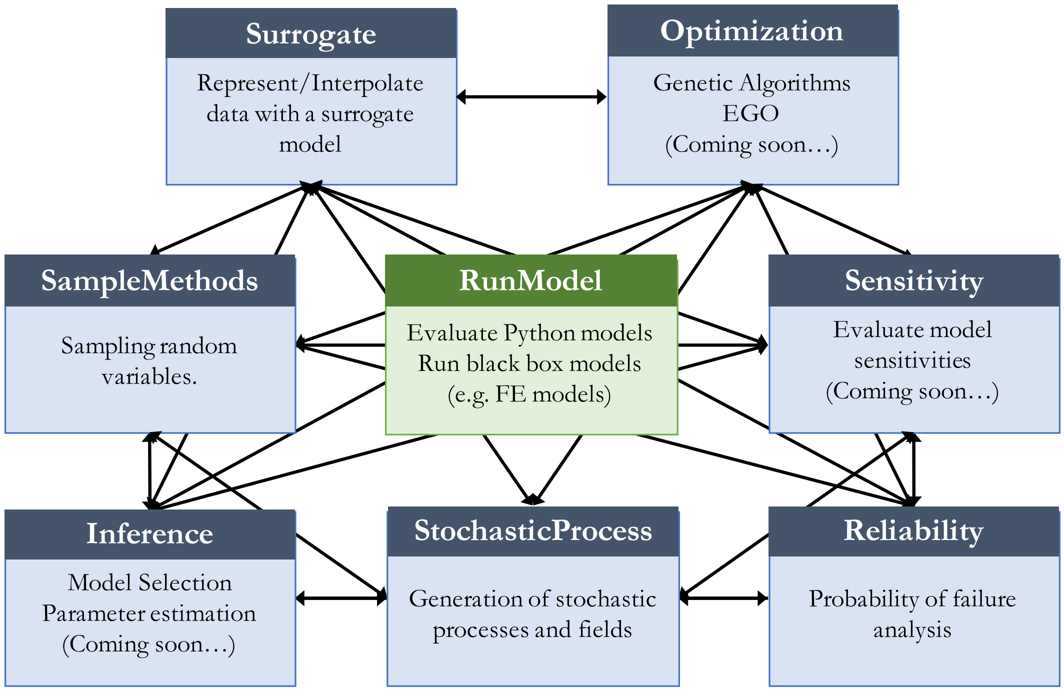

Figure 1: `UQpy` modules and their basic architecture.

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 pre-processing 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 4.6, is designed to interface with any user-defined third-party computational

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

Module	**Class**	**Description**	**Version**
SampleMethods	LSS	Latinized Stratified Sampling	2.0.0
	PSS	Partially Stratified Sampling	2.0.0
	LPSS	Latinized Partially Stratified Sampling	2.0.0
	IS	Importance Sampling	2.0.0
	RSS	Refined Stratified Sampling	3.0.0
	GE-RSS	Gradient Enhance Refined 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
	Simplex	Simplex Sampling	3.0.0
	Composition	Composition Sampling Method	2.0.0
	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
	Kriging	Gaussian Process/Kriging Surrogate	2.0.0
	MMK	Multimodel Kriging Surrogate	2.0.0
	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	TaylorSeries	Taylor Series for First Order Reliability Method and/or Second Order Reliability Method	2.0.0
	TRS	Targeted Random Sampling	3.0.0
	SESS	Surrogate Enhance Stochastic Search	3.0.0
	AK-MCS	Adaptive Kriging Monte Carlo Simulation	2.0.0
Inference	InfoModelSelection	Information Theoretic Model Selection	2.0.0
	BayesModelSelection	Bayesian Model Selection	2.0.0
	BayesParameter	Bayesian Parameter Estimation	2.0.0
	KDE	Kernel Density Estimation	2.0.0
Optimization	EGO	Efficient Global Optimization	2.0.0
	GA	Genetic Algorithms	3.0.0
Sensitivity	Sobol	Sobol Indices	2.0.0
	PCESobol	Polynomial Chaos Sobol Indices	3.0.0

model (either through user-defined shell scripts or a Python script) or directly with a Python model.

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.

Copyright ©2018 – Michael D. Shields

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the ”Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ”AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

4 UQpy Modules, Classes, & Functions

UQpy is structured in eight core modules (see Figure 1), each centered around specific functionalities. The modules are as follows:

1. Distributions: This module contains a set of supported distributions and their functions (pdf, cdf, moments, random numbers, fit, inverse cdf, log_pdf).

2. SampleMethods: This module contains a set of classes and functions 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.

3. Inference: (Coming in Version 2.0.0) 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 and functions designed specifically to estimate rare event probabilities and probability of failure.

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

6. Sensitivity: (Coming in Version 2.0.0) This module contains a set of classes and functions for performing global and local sensitivity analysis.

7. Optimization: (Coming in Version 2.0.0) This module contains a set of classes and functions to perform optimization for stochastic problems.

8. StochasticProcess: (Coming in Version 2.0.0) This module contains a set of classes and functions for the simulation of stochastic processes and fields.

9. RunModel: This module contains a set of classes and functions that allows UQpy to initiate simulations using Python or third-party computational solvers, and monitor and post-process simulation results.

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

4.1 SampleMethods Module

The `SampleMethods` module consists of classes and functions to draw samples from random variables, to induce or remove correlation from samples and to transform the samples. 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
`Correlate`	Induces correlation
`Decorrelate`	Removes correlation
`Nataf`	Nataf transformation
`InvNataf`	Inverse Nataf transformation

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.

4.1.1 UQpy.SampleMethods.MCS

`MCS` is a class for Monte Carlo Sampling – random sampling from independent random variables having user specified distributions. 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**
`dimension`	Input		⋆
`dist_name`	Input	⋆	
`dist_params`	Input	⋆	
`nsamples`	Input	⋆	
`samplesU01`	Output		
`samples`	Output		

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

MCS Class Attributes			
Attribute	**Type**	**Options**	**Default**
`dimension`	*integer*		`dimension = len(dist_name)`
`dist_name`	*function/string list*	See `Distributions` Module or User-defined function	
`dist_params`	*ndarray list*		
`nsamples`	*integer*		`None`
`samplesU01`	*ndarray*		
`samples`	*ndarray*		

Detailed Description of `MCS` Class Attributes:

Input Attributes:

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

- `dist_name`:
  Defines the name of the distribution 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. 5.1) or the 'custom_dist.py' (again see Sec. 5.1).

  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.

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.

- 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 nsamples $\times$ dimension containing the samples generated uniformly on the hypercube $[0, 1]^{\texttt{dimension}}$.

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

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

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

4.1.2 `UQpy.SampleMethods.LHS`

`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. 5.1) or the 'custom_dist.py' (again see Sec. 5.1).

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

4.1.3 `UQpy.SampleMethods.STS`

`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		⋆
`input_file`	Input		⋆
`samples`	Output		
`samplesU01`	Output		
`strata`	Input		

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`
`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.

- `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. 5.1) or the 'custom_dist.py' (again see Sec. 5.1).

  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 4.1.4.

`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.

- `input_file`:
  Specifies the file path of for a text file defining a stratification. See Section 4.1.4

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.

4.1.4 `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		$\star$
`input_file`	Input		$\star$
`origins`	Output		
`widths`	Output		
`weights`	Input		

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 4.1.3.

  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.

4.1.5 UQpy.SampleMethods.MCMC

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_type	Input		⋆
pdf_target	Input	⋆	
pdf_target_params	Input		⋆
algorithm	Input		⋆
jump	Input		⋆
nsamples	Input	⋆	
seed	Input		⋆
nburn	Input		⋆
samples	Output		

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

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

`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_type`	*string*	'marginal_pdf'
		'joint_pdf'
			if `algorithm` = 'MMH':
			`pdf_target_type` = 'marginal_pdf'
			if `algorithm`='Stretch':
			`pdf_target_type` = 'joint_pdf'

`pdf_target`	*function* *string*		Normal$(\mathbf{0}, \mathbf{I})$
`pdf_target_params`	*float* *float list*		`None`
`jump`	*integer*		1
`nsamples`	*integer*		`None`
`seed`	*ndarray* *ndarray list*		array(0,0,. . . ,0) size = 1 × `dimension`
`nburn`	*integer*		0
`samples`	*ndarray*		

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 [5, 4, 1].
  - 'MMH':
    Component-wise modified Metropolis-Hastings algorithm. For a description of the algorithm, see [1].
  - 'Stretch':
    Affine invariant ensemble sampler employing "stretch" moves. For a description of the algorithm, see [2].

- `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':
    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 [2].

  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.]

- `pdf_target`:
  Specifies the target probability density function from which to draw MCMC samples (i.e. the stationary distribution of the Markov chain). `pdf_target` must be passed into `MCMC` as a function. In `UQpy`, this can be achieved in two ways:

  - Direct function definition:
    The easiest way to define `pdf_target` is to create a function in the python script that calls `MCMC`. When the function is directly defined, `pdf_target` is specified directly using the function name (not as a string).
  - Definition through 'custom_pdf.py':
    If the function is to be called frequently by the user or may need to be shared among python scripts in a project, the user may define the function in a python script 'custom_pdf.py' that resides in the user's working directory. When this is the case, `pdf_target` is specified by a string that corresponds to the function name in 'custom_pdf.py'. See Section 5.1 for a detailed description of 'custom_pdf.py'.

In both cases, the function must be defined to accept two parameters:

1. The point at which to compute the pdf,
2. A list of parameters of the pdf specified through `pdf_target_params`

If the pdf does not have any user-defined parameters, the user still must define the function to accept a parameter list.

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 into the function defined by `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 skipping $n-1$ states between accepted states of the chain.

- `nsamples`
  Specifies the number of samples to be generated (not including 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 zeros with size $1 \times$ `dimension`.

  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. [2]. The default value in the table above is not valid for `algorithm` = 'Stretch'.

- `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'

Output Attributes:

- `samples`:
  The only output of the `MCMC` class are the generated samples. The samples are returned as a numpy array of dimension `nsamples` × `dimension`.

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.

4.1.6 `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 $\mathbf{C}$ denote the symmetric positive definite correlation matrix and $\mathbf{X}$ denote the `nsamples`× `dimension` array of independent standard normal samples. Perform the Cholesky decomposition such that:

$$\mathbf{C} = \mathbf{U}\mathbf{U}^T \tag{1}$$

where $\mathbf{U}$ is a lower-triangular matrix. The `nsamples`× `dimension` array, $\mathbf{Y}$ of correlated standard normal samples possessing correlation $\mathbf{C}$ is determined by:

$$\mathbf{Y}^T = \mathbf{U}\mathbf{X}^T \tag{2}$$

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` $\times$ `dimension` and satisfy:

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

  `0 < corr_norm[i, j] < 1` for `i` $\neq$ `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` $\times$ `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` $\times$ `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`.

4.1.7 `UQpy.SampleMethods.Decorrelate`

`Decorrelate` is a class for removing correlation from a `nsamples`×`dimension` array, $\mathbf{Y}$, of standard normal random samples with correlation matrix $\mathbf{C}$. This is performed by simply inverting the expression in Eq. (2) as:

$$\mathbf{X}^T = \mathbf{U}^{-1}\mathbf{Y}^T \tag{3}$$

to obtain the `nsamples`×`dimension` array, $\mathbf{X}$, 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` × `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` × `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.

4.1.8 `UQpy.SampleMethods.Nataf`

`Nataf` is a class for transforming standard normal random samples to a prescribed non-Gaussian distribution using the Nataf transform as follows. Let $\mathbf{X}$ denote an n-dimensional standard normal random vector and let $F_i(y), i = 1, \ldots, n$ be the marginal cumulative distribution functions of the n non-Gaussian random variables. The non-Gaussian random vector, $\mathbf{Y}$, following $F_i(y)$ is defined component-wise through the transformation:

$$Y_i = F_i^{-1}(\Phi(X_i)) \tag{4}$$

where $\Phi(x)$ is the standard normal cumulative distribution function.

When the random vector X has correlated components possessing correlation matrix $\mathbf{C}$ and correlation coefficients ρ_{ij} between components X_i and X_j, the transformation in Eq. (4) causes a so-called *correlation distortion* such that the correlation coefficient between the non-Gaussian variables Y_i and Y_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:

$$\xi_{ij} = \frac{1}{\sigma_{Y_i}\sigma_{Y_j}} \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \left(F_i^{-1}(\Phi(x_i)) - \mu_{Y_i}\right)\left(F_j^{-1}(\Phi(x_j)) - \mu_{Y_j}\right) \phi(x_i, x_j; \rho_{ij}) dx_i dx_j \tag{5}$$

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		$\star$
`corr_norm`	Input	$\star$	
`dist_name`	Input	$\star$	
`dist_params`	Input	$\star$	
`dimension`	Input	$\star$	$\star$
`samplesN01`	Output		
`samples`	Output		
`corr`	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*	`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 5.1	
`dist_params`	*ndarray list*	See Section 5.1	
`samplesN01`	*ndarray*		
`samples`	*ndarray*		
`corr`	*ndarray*		
`jacobian`	*ndarray list*		

Detailed Description of `Nataf` 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`$\times$ `dimension` `numpy` array. The 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 `Nataf` 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 `Nataf` calculates the correlation distortion of the standard normal correlation matrix `corr_norm` from Eq. (5).

  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` $\times$ `dimension` and satisfy:

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

  `0 < corr_norm[i, j] < 1` for `i` $\neq$ `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`$\times$`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. 5.1). 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. 5.1).

  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. 5.1).

  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 thave 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` × `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.

- Nataf - Example 1.ipynb:
  In this example, the `Nataf` 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.

- Nataf - Example 2.ipynb:
  In this example, the `Nataf` 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.

- Nataf - Example 3.ipynb:
  In this example, the `Nataf` class is used to calculate the correlation distortion for the transformation of two correlated random variables from a standard normal to a lognormal distribution.

4.1.9 `UQpy.SampleMethods.InvNataf`

`InvNataf` is a class for transforming non-Gaussian random variables to equivalent standard normal space. 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	$\star$	$\star$
`dimension`	Input	$\star$	$\star$
`corr`	Input	$\star$	
`dist_name`	Input	$\star$	$\star$
`dist_params`	Input	$\star$	$\star$
`samplesNG`	Output		
`samples`	Output		
`corr_norm`	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*	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 InvNataf 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, Nataf` 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 `InvNataf` 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` $\times$ `dimension` and satisfy:

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

  `corr[i, j] < 1` for `i` $\neq$ `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. 5.1) or the 'custom_dist.py' (again see Sec. 5.1).

  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, Nataf` 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, Nataf` 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, Nataf` 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.

- InvNataf - Example 1.ipynb:
  In this example, `InvNataf` class is used in order to transform 2 correlated lognormal variables to two standard normal random variables.
- InvNataf - Example 2.ipynb:
  In this example, `Nataf` class is used to perform the Iterative Translation Approximation Method (ITAM) [6] to estimate the underlying Gaussian correlation from known values of the correlation for lognormal random variables.

4.2 `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

4.2.1 `UQpy.Surrogates.SROM`

`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 [3]. 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. 5.1) or the cdf defined by 'custom_dist.py' (again see Sec. 5.1).

  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 $1 \times$ `dimension`: If error in either, but not both, first or second-order moments is included in SROM.
  - Array of size $2 \times$ `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` $\times$ `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 [3]. 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 [3].

- `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 [3].
  - 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 [3].
  - 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 [3].

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`.

4.2.2 `UQpy.Surrogates.Kriging` (Coming in V2.0)

4.3 `Reliability` Module

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:

```
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.

4.3.1 `UQpy.Reliability.SubsetSimulation`

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		⋆
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 [1].
  - 'Stretch':
    Affine invariant ensemble sampler employing "stretch" moves. For a description of the algorithm, see [2].

  `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 4.1.5

- `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 4.1.5

- `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 4.1.5

- `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 4.1.5

- `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 4.1.5

- `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 4.6.

- `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 4.6.

  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 4.6.

- `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 4.6.

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 4.6.

- `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.

4.3.2 `UQpy.Reliability.TaylorSeries` (Coming in V2.0)

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

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

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

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

4.4 `Inference` Module

4.4.1 `InfoModelSelection` (Coming in V2.0)

Information-theoretic model selection coming soon...

4.4.2 `BayesModelSelection` (Coming in V2.0)

Bayesian model selection coming soon...

4.4.3 `BayesParameterEstimation` (Coming in V2.0)

Bayesian parameter estimation coming soon...

4.5 `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.

4.5.1 `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	$\star$	
`S`	Input	$\star$	
`dw`	Input	$\star$	
`nt`	Input	$\star$	
`nw`	Input	$\star$	
`case`	Input	$\star$	
`g`	Input	$\star$	
`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.

4.5.2 `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:

```python
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	$\star$	
S	Input	$\star$	
B	Input	$\star$	
dt	Input	$\star$	
dw	Input	$\star$	
nt	Input	$\star$	
nw	Input	$\star$	
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.

4.5.3 `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	$\star$	
`R`	Input	$\star$	
`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.

4.5.4 `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.

4.5.5 `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	$\star$	
`R_ng`	Input	$\star$	
`marginal`	Input	$\star$	
`params`	Input	$\star$	
`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.

4.6 `RunModel` Module

The `RunModel` module is how `UQpy` calls user-defined computational models and collects the results from the output of those simulations. Using the `RunModel` module requires the user to be familiar with either shell scripting or python scripting. 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 attributes of the `RunModel` class are listed below:

`RunModel` Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
`dimension`	Input		$\star$
`samples`	Input		$\star$
`model_type`	Input		$\star$
`model_script`	Input	$\star$	
`input_script`	Input		$\star$
`output_script`	Input		$\star$
`cpu`	Input		$\star$
`model_eval`	Output		

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

RunModel Class Attributes			
Attribute	**Type**	**Options**	**Default**
`dimension`	*integer*		`dimension` = 1
`samples`	*nparray*		`None`
`model_type`	*string*	'python' `None`	`None`
`model_script`	*string*	Must be '.py' or '.sh'	
`input_script`	*string*	Must be '.py' or '.sh'	
`output_script`	*string*	Must be '.py' or '.sh'	
`cpu`	*integer*	`cpu` $<$ # of available CPUs	`cpu = 1`

`model_eval`	*class object*	`RunPythonModel`
		`RunSerial`
		`RunParallel`	

Detailed Description of `RunModel` Class Attributes:

Input Attributes:

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

- `samples`
  Specifies the sample points at which to evaluate the model.

  If `samples` is not specified, `RunModel` will search the working directory for a file called 'UQpy_Samples.txt'. Creating this text file allows an alternate way of defining samples for the `RunModel` class that does not require the samples to be generated by `UQpy`. Formatting specifications for 'UQpy_Samples.txt' are given in Section 4.6.3.

- `model_type`
  Specifies the type of model that will be evaluated.

  If `model_type` = 'python', then the model is either a user-defined Python model (i.e. a solver written in Python) or the model is a third-party model with both pre- and post-processing handled by a single Python script. Using a Python model or a Python script to invoke the model allows `UQpy` to handle message passing internally in Python. This mode of operation requires the definition of only one script, defined by `model_script`, which must be a .py file. For more details, see Section 4.6.1.

If `model_type = None`, then the model is called through a series of either shell or Python scripts. This is a more general framework that relies on text files to pass `samples` into the model input file and to retrieve the model quantity of interest (defined by `RunModel.model_eval.QOI`. This mode of operation requires the user to define three scripts:

1. `input_script`: This user-defined script (which may be a .sh or .py file), reads a text file of samples generated from `UQpy` in a specified format (see Section 4.6.2) and generates input files for the computational model.
2. `model_script`: This user-defined script (which may be a .sh or .py file), calls the computational model and initiates the simulations.
3. `output_script`: This user-defined script (which may be a .sh or .py file), reads an output file from the computational model, extracts the desired quantity of interest, and prints the value(s) of this quantity of interest to a text file of specified format (see Section 4.6.2) that `UQpy` reads.

- `model_script`
  Specifies the user-defined script used to call the computational model. If `model_type = None`, `model_script` may be either a .py or .sh file. If `model_type` = 'python', `model_script` must be a .py file.

- `input_script`:
  Only used with `model_type = None`.

  Specifies the user-defined script used to read a text file containing a sample value with specified format and create an input file for the computational model. May be a .sh or .py file. See Section 4.6.2.

- `output_script`:
  Only used with `model_type = None`.

  Specifies the user-defined script used to read a model output file, extract the quantity of interest, and create a text file containing the quantity of interest in a specified format that can be ready by `UQpy`. May be a .sh or .py file. See Section 4.6.2.

- `cpu`:
  Specifies the number of CPUs over which to distribute the simulations.

This number must be less than the number of available CPUs on the computer performing the simulations.

Output Attributes:

- `model_eval`:
  This is an instance of one of three classes used to call the computational model.

  If `model_type` = 'python', `model_eval` is an instance of the `RunPythonModel` class defined in the Python `model_script`. See Section 4.6.1.

  If `model_type = None`, `model_eval` is an instance either the `RunSerial` or `RunParallel` class, depending on whether the user specified serial (`cpu` = 1) or parallel (`cpu` > 1) computing. See Section 4.6.2.

`RunModel` Workflows

There are two general workflows for the `RunModel` class. In the first, a model is defined or called through python scripts, which allows all sample passing to be performed internally and therefore has less computational "overhead." In the second workflow, samples and solutions are passed between `UQpy` and a third-party solver through text files. The following sections detail these two workflows.

4.6.1 `RunModel` with direct Python communications (`model_type` = 'python')

The fastest, simplest, and preferred way to run a model using `UQpy` is by linking `UQpy` to a Python script that calls or runs the model. This link occurs by calling the `RunModel` class, setting `model_type` = 'python', and pointing it to the user-defined Python script that will execute the model. `RunModel` is pointed to the Python script by defining the input parameter `model_script` as a string having the name of the Python script (note this file must be a .py file). More details on defining `model_script` can be found in Section ??. Figure 2 shows a general flow-chart for the `RunModel` class invoking a Python script to run simulations.

`UQpy` calls the Python script defined by `model_script` through the class `RunPythonModel`, which must be present in `model_script` and is defined as follows:

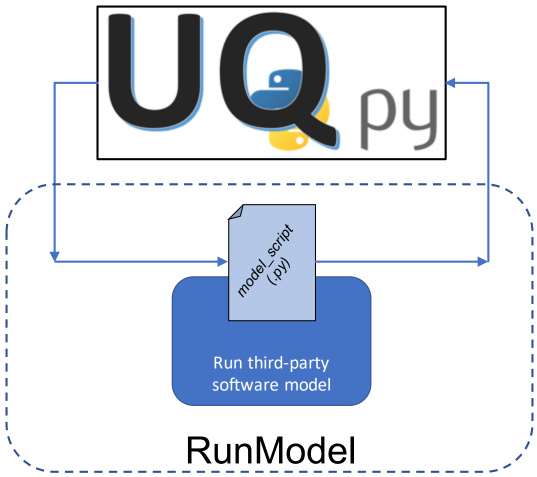

Figure 2: General workflow for running a model from a python script (`model_type` = ‘pthon’) using the `RunModel` class of `UQpy`.

```
class RunPythonModel:

    def __init__(self, samples=None, dimension=None):

    self.samples = samples
    self.dimension = dimension
    self.QOI = list()
```

The `RunPythonModel` class in `model_script` must accept, as input, a set of samples and the dimension of the samples and return, as output, a list containing the quantity of interest (`self.QOI`) computed for each sample. The attributes of the `RunPythonModel` are described below. Beyond these minimal requirements, the user has complete freedom to perform whatever operations she/he desires. That is, `model_script` may be used directly to perform some operations on the samples (e.g. solve a set of differential equations having parameters defined by the samples) or to pass the samples to input files and call a third-party model (e.g. Matlab, Abaqus, or a custom simulation code).

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

`RunPythonModel` Class Attribute Definitions			
Attribute	**Input/Output**	**Required**	**Optional**
`dimension`	Input	⋆	
`samples`	Input	⋆	
`QOI`	Output	⋆	

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

`RunPythonModel` Class Attributes			
Attribute	**Type**	**Options**	**Default**
`dimension`	*integer*		
`samples`	*nparray*		
`QOI`	*list*		

Detailed Description of `RunPythonModel` Class Attributes:

Input Attributes:

- `dimension`:
  A scalar integer value defining the dimension of the random variables.
- `samples`
  Specifies the sample points at which to evaluate the model.

Output Attributes:

- `QOI`:
  A list containing the quantity of interest returned from the model. Each item of the list corresponds to an associated sample value and may be of arbitrary data type.

Examples:
An example illustrating the use of the `RunModel` class with `model_type` = 'python' is provided in the following Jupyter script.

- Run_Python_Model.ipynb:
  In this example, the component-wise modified Metropolis-Hasting algorithm for MCMC is used to generate 15 (approximately) independent samples from a two-dimensional Rosenbrock pdf. The Rosenbrock pdf is defined as a function directly in the script. The samples are then passed to a Python model (python_model.py) that evaluates the sum of the components of each sample and returns the sum as the quantity of interest (`x.model_eval.QOI`).

Running a model in Python is strongly preferred both from the perspective of flexibility for the user, but also because it alleviates the burden of file passing as a means of communication between `UQpy` and model input/output. This is the topic of the next section.

4.6.2 `RunModel` with file passing communications (`model_type = None`)

The `RunModel` class supports an alternate means of running a model for users who prefer shell scripting or who prefer a more prescriptive workflow. This alternate means of running uses a set of scripts and text files to pass information from `UQpy` to a third-party model and return the results. This method of running the model supports both serial computation and parallel processing across multiple cores. It does not currently support distributed processing across multiple nodes in an HPC.

Figure 3 illustrates this workflow, which follows a three-step process:

1. Convert text files of `UQpy` samples to model input files.

2. Run the computational model.

3. Convert model output from each simulation to text files that can be read by `UQpy`.

This three step process is detailed in the following.

Step 1: For each sample value, `UQpy` generates a text file called 'UQpy_run_n.txt' where n indexes the sample number as illustrated in Figure 4. The user must pass the name of a shell or Python script (as a string through `input_script`) that reads 'UQpy_run_n.txt' and inserts the samples into an input file for the computational model. For specification of the formatting of 'UQpy_run_n.txt', see Section 4.6.3. An example `input_script` is provided in the example 'Matlab_Model_Serial.ipynb' provided below.

Step 2: For each sample value, a model input file is generated in step 1. `UQpy` then calls the user-defined `model_script` to run the computational model as illustrated in Figure 5. `RunModel` loops over all samples to run the model for each generated input file. This can be done either serially or in parallel over multiple processors. See description below.

Step 3: For each simulation, an output file is generated. The user-defined `output_script` is used to post-process these outputs, extract the desired

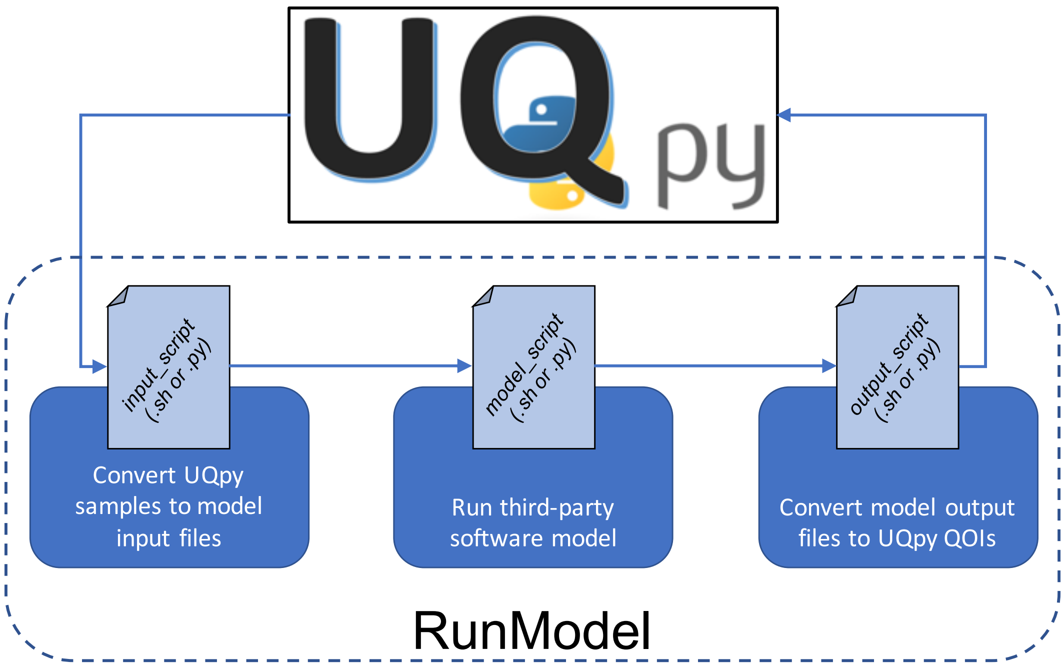

Figure 3: General workflow for running a third-party model with `UQpy` with samples and solutions passed through text files (`model_type = None`).

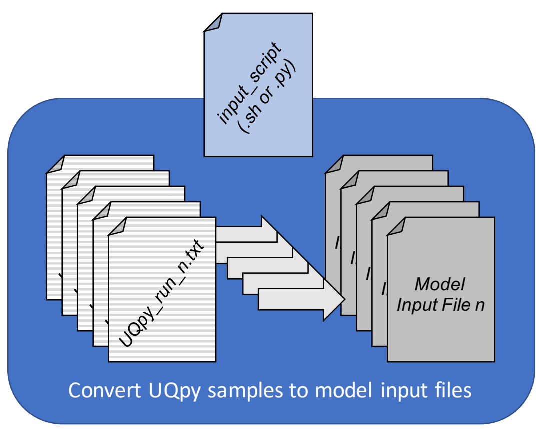

Figure 4: The user-defined `input_script` is used to read `UQpy` samples from text files defined as 'UQpy_run_n.txt' and create model input files.

quantity of interest, and write this quantity of interest to a text file named 'UQpy_eval_n.txt' where, again n indexes over the sample number as illustrated in Figure 6. For formatting specifications of 'UQpy_eval_n.txt', see

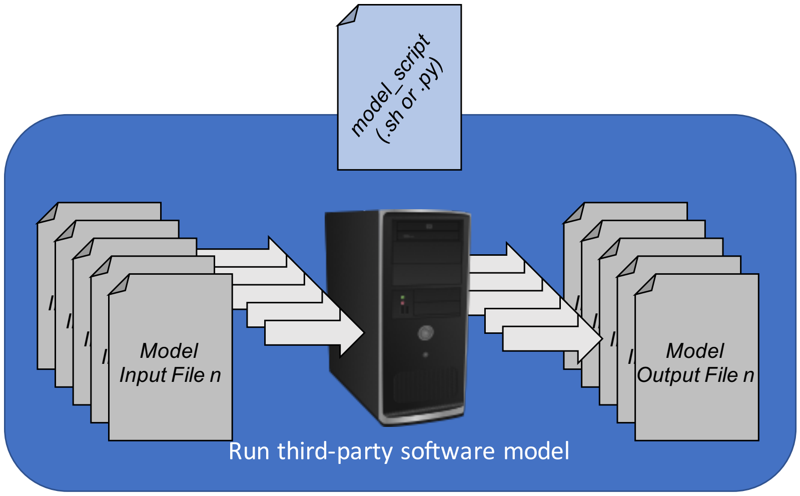

Figure 5: The user-defined `model_script` is used to run a third party software model using the model input files generated by the `input_script`. `UQpy` runs the model in a loop to evaluate all samples.

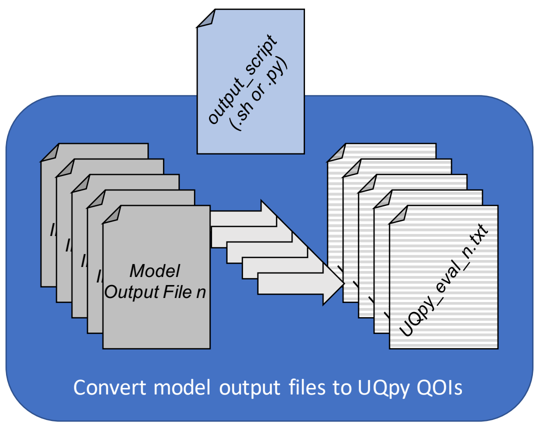

Figure 6: The user-defined `output_script` is used to post-process model results, extract a quantity of interest, and write that quantity of interest to 'UQpy_eval_n.txt' which can be read by `UQpy`.

Section 4.6.3.

`RunSerial` and `RunParallel`

Depending on the number of CPUs the user specifies via the `cpu` attribute, the model will either be run serially or in parallel across the specified number of CPUs by invoking the `RunSerial` and `RunParallel` sub-classes respectively.

When `cpu` $= 1$, the model is run by calling `RunSerial`, setting the instance of this class as `model_eval`, and returning the quantities of interest for the solution as `model_eval.QOI`.

When `cpu` > 1, the model is run by calling `RunParallel`, setting the instance of this class as `model_eval`, and returning the quantities of interest for the solution as `model_eval.QOI`. Given N samples, `RunParallel` bundles the N calculations into $\lfloor N/\texttt{cpu} \rfloor + \text{mod}\{N/\texttt{cpu}\}$ calculations on the first $\text{mod}\{N/\texttt{cpu}\}$ CPUs and $\lfloor N/\texttt{cpu} \rfloor$ calculations on all remaining CPUs.

Directory structure during model evaluation

To execute `RunModel`, the working directory must contain the necessary scripts (defined by `model_script`, `input_script`, and `output_script`) along with any other files necessary for model evaluation. These may include, among other things, a template model input file (to be edited by `input_script` to input sample values), compiled executable files for third-party software that runs locally, and/or 'UQpy_samples.txt' if samples are not being generated by `UQpy`. To avoid cluttering the working directory, the first step in model evaluation using `RunModel` is to create a new directory called 'tmp' and copy all files into this directory as illustrated in Figure 7.

From the 'tmp' directory, the appropriate class `RunSerial` or `RunParallel` is executed. The first step in either process is to generate, from the samples (defined either by `RunModel.samples` or 'UQpy_Samples.txt'), a single text file 'UQpy_run_n.txt' where n indexes the sample number, for each sample value. These are the files that are read by `input_script`. The model evaluation process then proceeds as illustrated in Figures 3 - 6, ending with the quantities of interest returned in text files 'UQpy_eval_n.txt' and also saved internally within `RunModel` as `RunModel.model_eval.QOI`.

The final step is to clean up the working directory. As illustrated in Figure 8, the input files are returned to the original working directory, all output files 'UQpy_eval_n.txt' are moved to a new directory 'UQpyOut', and the 'tmp' directory is removed.

Examples:

Two examples illustrating the use of the `RunModel` class with `model_type = None`' are provided that run a simple Matlab model from two-dimensional input in the following Jupyter scripts.

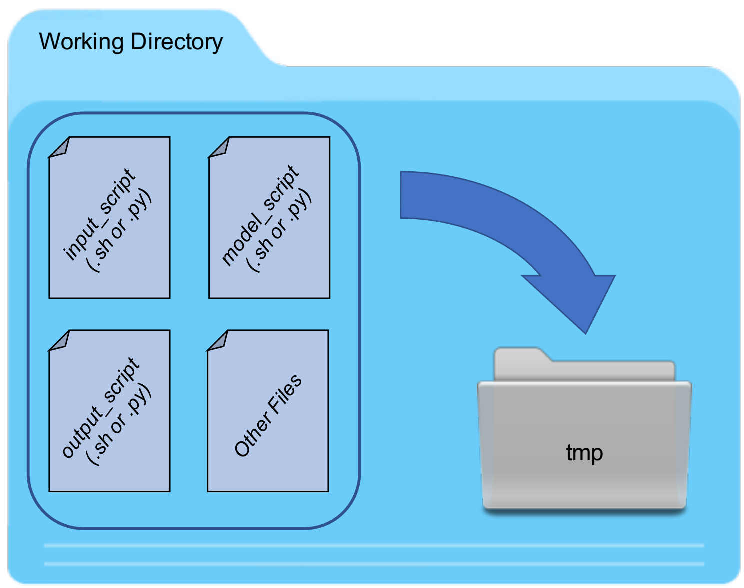

Figure 7: The first step in executing `RunModel` is to copy all files into a temporary subdirectory of the working directory called 'tmp' where all computations will be performed.

- Run_Serial_Matlab_Model.ipynb:
  In this example, the component-wise modified Metropolis-Hasting algorithm for MCMC is used to generate 15 (approximately) independent samples from a two-dimensional Rosenbrock pdf. The Rosenbrock pdf is defined as a function directly in the script. The samples are then saved as a text file 'UQpy_Samples.txt' to illustrate that `RunModel` can read samples from a text file. A simple Matlab model 'matlab_model.m' is included that evaluates the sum of the components of each sample and returns them as the as the quantity of interest (`x.model_eval.QOI`) and saves each sum as a text file 'UQpy_eval_n', n = 1,...,15 in the folder 'UQpyOut'. The `RunModel` class is run serially, `cpu` = 1, meaning that all 15 Matlab calculations are performed sequentially. Finally, the resulting data structures are printed to illustrated how `UQpy` saves model output.

- Run_Parallel_Matlab_Model.ipynb:
  In this example, the component-wise modified Metropolis-Hasting algorithm for MCMC is used to generate 15 (approximately) independent

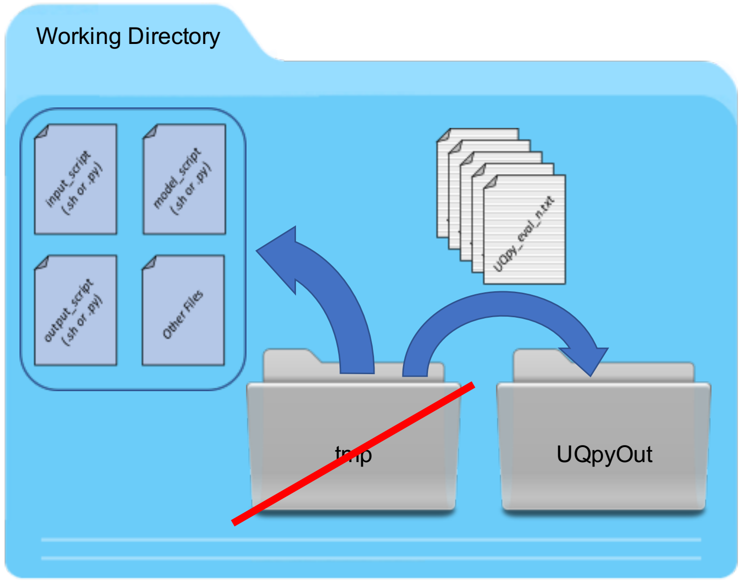

Figure 8: Final cleanup of the working director is the last step of model evaluation using `RunModel`. In the process, the input files are returned to the original working directory, all output files 'UQpy_eval_n.txt' are moved to a directory 'UQpyOut', and the 'tmp' directory is removed.

samples from a two-dimensional Rosenbrock pdf. The Rosenbrock pdf is defined as a function directly in the script. The samples are passed directly into the `RunModel` class. A simple Matlab model 'matlab_model.m' is included that evaluates the sum of the components of each sample and returns them as the as the quantity of interest (`x.model_eval.QOI`) and saves each sum as a text file 'UQpy_eval_n', n = 1,...,15 in the folder 'UQpyOut'. The `RunModel` class is run in parallel over four CPUs, `cpu` = 4. The 15 Matlab calculations bundled into groups of 4, 4, 4, and 3 calculations and each group is performed sequentially over one assigned CPUs. Finally, the resulting data structures are printed to illustrated how `UQpy` saves model output.

4.6.3 Files and scripts used by `RunModel`

As discussed in the sections above and illustrated in the examples, the `RunModel` class utilizes a number of files and scripts in order to execute the computational model. This section is intended to provide a closer look at each of these files, their structure, and when/if they are required.

- 'UQpy_Samples.txt':
  This user-defined text file allows the user to pass samples into the `RunModel` class without drawing new samples from `UQpy`. Examples of when this file may be used include, but are not limited to, the following cases:

  - The user generates a set of samples using another package (not `UQpy`), but still wishes to use `UQpy` as the driver to run the model.
  - The user wishes to retain the same set of samples when evaluating a model that changes in some way. For example, running models of different mesh resolution with the same input values.

  File Format: 'UQpy_Samples.txt' is an ASCII formatted text file having one sample per line with whitespace delimiters separating each component of the samples.

  'UQpy_Samples.txt' can be used with `model_type = None` and `model_type` = 'python'.

- 'UQpy_run_n.txt':
  Each 'UQpy_run_n.txt' (where n indexes the sample number) is a `UQpy` defined ASCII text file containing a single sample. While the user is not required to generate this file, it is important that the user know its format as the user-defined `input_script` must read this file and place its sample values into the model input file.

  File Format: 'UQpy_run_n.txt' is an ASCII formatted text file having one sample with whitespace delimiters separating each component of the sample.

  These files are generated only when using `RunModel` with `model_type = None`.

- 'UQpy_eval_n.txt':
  Each 'UQpy_eval_n.txt' (where n indexes the sample number) is a user-created ASCII text file containing a single quantity of interest generated from post-processing the model output file from the n^{th} simulation. The

user must generate this file using `output_script` so it is important that the user know its format.

File Format: 'UQpy_eval_n.txt' is an ASCII formatted text file having one quantity of interest with whitespace delimiters separating each component of the quantity of interest (if it is vector-valued). If the quantity of interest is matrix-valued or tensor-valued, it currently must be unpacked into a vector for saving in 'UQpy_eval_n.txt'. This will change in the future.

These files need to be generated only when using `RunModel` with `model_type = None`.

- `input_script`: `input_script` is a script that reads each sample in 'UQpy_run_n.txt' and places the values in the appropriate location in the model input file.

  File Format: `input_script` must be a python script (.py) or shell script (.sh).

  `input_script` is used only when using `RunModel` with `model_type = None`.

- `model_script`: `model_script` is the user-defined script that runs the computational model. It can be employed in two different ways depending on the assignment of `model_type`.

  – `model_type = None`: `model_script` is responsible only for initializing the computational model.

    File Format: `model_script` must be a python script (.py) or shell script (.sh).

  – `model_type` = 'python': `model_script` may contain the computational model itself. In such case, the samples that are passed into `Runmodel` are input directly into the python solver. `model_script` may also call an external solver. In this case, `model_script` must also place the sample values in the model input file and post-process the model output to generate `model_eval.QOI`.

    File Format: `model_script` must be a python script (.py) containing the `RunPythonModel` class as discussed in Section 4.6.1.

- `output_script` `output_script` is the user-defined script that post-processes the model output to extract the user-specified quantity of interest and write this quantity of interest to the 'UQpy_eval_n.txt' files.

  File Format: `model_script` must be a python script (.py) or shell script (.sh).

  `output_script` is used only when using `RunModel` with `model_type = None`.

- Model Input file The model input file is a user-defined file that is also specific to the model application. The model input file is typically a standard format file that defines all deterministic parameters, geometry, material, properties, etc. of the computational model. This file should also have place-holders for the input of sample values generated by `UQpy`. In the future, these place-holders will be standardized, but as yet they are not.

- Executable Software Often, the working directory will contain an executable software program. When this software is user-defined (as may be the case for custom solvers), the executable program may need to reside in the current working directory.

4.6.4 Template scripts for common software applications

- Matlab
  Coming soon...

- Abaqus
  Coming soon...

- OpenSEAS
  Coming soon...

- OpenFOAM
  Coming soon...

- FEAP
  Coming soon...

- SAFIR
  Coming soon...

5 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.

5.1 Distributions Module

The Distributions module performs probability distribution related operations. 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:

```
from UQpy import Distributions
```

The Distributions module contains a single class, the Distribution class, possessing the following attributes:

Distribution Class Attribute Definitions			
Attribute	Input/Output	**Type**	**Required**
name	Input	*string*	*
params	Input	*list*	*
pdf	Output	*function*	
rvs	Output	*function*	
cdf	Output	*function*	
icdf	Output	*function*	
log_pdf	Output	*function*	
fit	Output	*function*	
moments	Output	*function*	

With the exception of the custom distribution, the Distribution 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 can be found in the table below:

`Distribution` Class Attributes			
Attribute	**Type**	**Options**	**Default**
`name`	*string*	See list below.	
`params`	*list*		
`pdf`	*function*		
`rvs`	*function*		
`cdf`	*function*		
`icdf`	*function*		
`log_pdf`	*function*		
`fit`	*function*		
`moments`	*function*		

Detailed Description of `Distribution` Class Attributes:

Input Attributes:

- `name`:
  A string designating the distribution name. Available distributions are shown in the table below.

  `name` must be specified. `Distribution` does not have a default distribution type.

- `params`:
  Defines the parameters of the distribution for each random variable as a list. Parameters for all available distributions are shown in the table below. Generally, the parameters adhere to the defined parameters in `Scipy.stats`.

  `params` must be specified. There are no default parameter values for any distribution.

Output Attributes:

- `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.

  The function is called as follows:

  ```
  Distribution.pdf(x,params)
  ```

- `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.

  The function is called as follows:

  ```
  Distribution.rvs(params, 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.

  The function is called as follows:

  ```
  Distribution.cdf(x,params)
  ```

- `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.

  The function is called as follows:

  ```
  Distribution.icdf(x,params)
  ```

- `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.

  The function is called as follows:

  ```
  Distribution.log_pdf(x,params)
  ```

- `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`.

The function is called as follows:

```
Distribution.fit(y)
```

- `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.

  The function is called as follows:

  ```
  Distribution.moments(params)
  ```

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’	$[\sigma, \mu]$ $s = \sigma,\ scale = \exp(\mu)$ $\sigma > 0$
`Maxwell-Boltzmann`	‘maxwell’	$[loc, scale]$ $scale > 0$
`Normal(Gaussian)`	‘normal’ or ‘gaussian’	$[\mu, \sigma]$ $loc = \mu,\ scale = \sigma$ $\sigma > 0$
`Pareto`	‘pareto’	$[b, loc, scale]$ $b, scale > 0$
`Rayleigh`	‘rayleigh’	$[loc, scale]$ $scale > 0$
`Uniform`	‘uniform’	$[a, b]$ $loc = a,\ scale = b - a$ $b > a$

Custom Distributions:

Other distributions can be easily added by defining the appropriate functions in `custom_dist.py`. These functions are those listed in the “`Distribution`

Class Attributes" table above.

Description of `custom_dist.py`
The script `custom_dist.py` allows the user to define a custom probability distribution function. In the script, the user may define 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. For compatibility with `UQpy`, the name of each function, `func_name`, 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.

5.2 `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 `Utilities`	
Name	**Description**
`transform_ng_to_g`	Transform non-Gaussian to Gaussian rvs
`transform_g_to_ng`	Transform Gaussian to non-Gaussian rvs
`itam`	Iterative Translation Approximation Method
`run_corr`	Correlates standard normal variables
`run_decorr`	Decorrelates standard normal variables
`correlation_distortion`	Evaluate the modified correlation matrix
`bi_variate_normal_pdf`	Evaluate the values of the bi-variate normal pdf
`_get_a_plus`	A supporting function for the `nearest_pd function`
`_get_ps`	A supporting function for the `nearest_pd function`
`_get_pu`	A supporting function for the `nearest_pd function`
`nearest_psd`	Compute the nearest positive semi definite matrix
`nearest_pd`	Find the nearest positive-definite matrix
`estimate_psd`	Estimate the Power Spectrum given an ensemble of samples
`s_to_r`	Transform the power spectrum to an autocorrelation function
`r_to_s`	Transform the autocorrelation function to a power spectrum
`is_pd`	Returns true when input is positive-definite.

6 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] 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.

[2] Jonathan Goodman and Jonathan Weare. Ensemble samplers with affine invariance. *Communications in applied mathematics and computational science*, 5(1):65–80, 2010.

[3] M. Grigoriu. Reduced order models for random functions. Application to stochastic problems. *Applied Mathematical Modelling*, 33(1):161–175, 2009.

[4] W K Hastings. Monte Carlo Sampling Methods Using Markov Chains and Their Applications. *Biometrika*, 57(1):97–109, 1970.

[5] 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.

[6] 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.