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 1.0.0

[*]michael.shields@jhu.edu
[†]dgiovan1@jhu.edu

Contents

1 Overview

UQpy (Uncertainty Quantification with python) is a general purpose Python toolbox for modeling uncertainty in the simulation of physical and mathematical systems. The code is organized as a set of modules centered around core capabilities in Uncertainty Quantification (UQ) as illustrated in Figure 1. The modules are distinct, but are designed to be easily extensible (new capabilities can be easily added and integrated into the code, see Section 6) and to easily call one another.

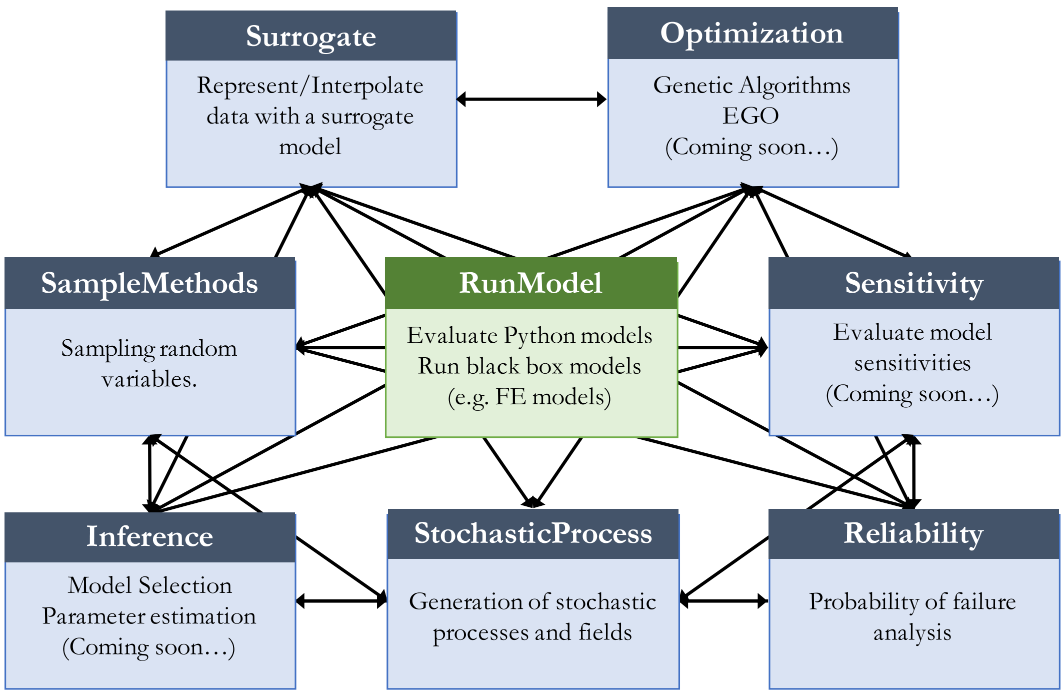

Figure 1: UQpy modules and their basic architecture.

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. 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

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

Module	**Class**	**Description**	**Version**
`SampleMethods`	`MCS`	Monte Carlo Sampling	1.0.0
	`LHS`	Latin Hypercube Sampling	1.0.0
	`STS`	Stratified Sampling	1.0.0
	`MCMC`	Markov Chain Monte Carlo	1.0.0
`Surrogates`	`SROM`	Stochastic Reduced Order Model	1.0.0
`Reliability`	`SubsetSimulation`	Subset Simulation	1.0.0

its module-class structure, the various classes can easily invoke one-another and can be combined in any way the user desires. A simple example of this is that the `SubsetSimulation` class in the `Reliability` module invokes the `MCMC` class from the `SampleMethods` module.

The various classes and modules interface in a straightforward manner with computational models of physical or mathematical systems through the `RunModel` module shown in the center of the chart in Figure 1. The `RunModel` module allows `UQpy` to serve not just as a useful tool for performing UQ operations, but also as the driver for a complete uncertainty study - including preprocessing operations, submission and execution of computational model evaluations, and monitoring and post-processing of results. Thus, it is amenable to performing adaptivity UQ anayses. The `RunModel` module, detailed in Section 5.8, is designed to interface with any user-defined third-party computational model (either through user-defined shell scripts or a Python script) or directly with a Python model.

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

Module	Class	Description	Version
SampleMethods	LSS	Latinized Stratified Sampling	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
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	FORM	First Order Reliability Method	2.0.0
	SORM	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	AIC	Akaike Information Criterion	2.0.0
	BIC	Bayesian Information Criterion	2.0.0
	ModelProbability	Model Probability	2.0.0
	Evidence	Bayesian Model Evidence	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
StochasticProcess	SRM	Spectral Representation Method	2.0.0
	KL	Karhunen-Loeve Expansion	2.0.0
	BSRM	Bispectral Representation Method	2.0.0
	Translation	Translation Process	2.0.0
	ITAM	Iterative Translation Approximation Method	2.0.0

2 Installing UQpy

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

```
pip install UQpy
```

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

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

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

```
pip uninstall UQpy
```

2.1 Manual Installation

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

```
git clone https://github.com/SURGroup/UQpy.git
cd UQpy/
python setup.py install
```

Direct installation from GitHub is equivalent to pip installation.

`UQpy` can be uninstalled using pip as:

```
pip uninstall UQpy
```

2.2 Developer Installation

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

```
git clone https://github.com/SURGroup/UQpy.git
cd UQpy/
python setup.py develop
```

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

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

3 Compiled version of `UQpy`

A compiled version of `UQpy` is currently under development and is expected to be included with release 2.0.0. The compiled version will not require Python to be installed on the computer and will operate through text-based input files.

The compiled version will be available as a Microsoft application and a Windows executable.

4 License

UQpy is distributed under the MIT license.

5 `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. `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.

2. `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.

3. `Reliability`: This module contains a set of classes and functions designed specifically to estimate rare event probabilities and probability of failure.

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

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

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

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

8. `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.

5.1 `SampleMethods` Module

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

```
from UQpy import SampleMethods
```

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

Class	Method
`MCS`	Monte Carlo Sampling
`LHS`	Latin Hypercube Sampling
`STS`	Stratified Sampling
`MCMC`	Markov Chain Monte Carlo

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

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

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

5.1.1 `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		$\star$
`icdf`	Input	$\star$	
`icdf_params`	Input	$\star$	
`nsamples`	Input	$\star$	
`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(icdf)`
`icdf`	*function/string list*	See `Distributions` Module or User-defined function	
`icdf_params`	*ndarray list*		
`nsamples`	*integer*		`None`
`samplesU01`	*nparray*		
`samples`	*nparray*		

Detailed Description of `MCS` Class Attributes:

Input Attributes:

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

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

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

  If `icdf[i]` is a string, the distribution is matched with its corresponding inverse cdf (`inv_cdf`) in the `Distributions` module (see Sec. 5.9.1) or the inverse cdf defined by 'custom_dist.py' (again see Sec. 5.9.1).

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

icdf can contain an arbitrary combination of strings and functions.

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

icdf must be specified. There is no default value.

- icdf_params:
  Specifies the parameters for each inverse cdf in icdf.

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

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

  icdf_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.

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

5.1.2 `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		⋆
`icdf`	Input	⋆	
`icdf_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(icdf)`
`icdf`	*function/string list*	See `Distributions` Module or User-defined function	
`icdf_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.

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

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

  If `icdf[i]` is a string, the distribution is matched with its corresponding inverse cdf (`inv_cdf`) in the `Distributions` module (see Sec. 5.9.1) or the inverse cdf defined by 'custom_dist.py' (again see Sec. 5.9.1).

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

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

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

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

- `icdf_params`:
  Specifies the parameters for each inverse cdf in `icdf`.

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

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

  `icdf_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.

5.1.3 `UQpy.SampleMethods.STS`

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

```python
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		$\star$
`icdf`	Input	$\star$	
`icdf_params`	Input	$\star$	
`sts_design`	Input		$\star$
`input_file`	Input		$\star$
`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)`
`icdf`	*function/string list*	See `Distributions` Module or User-defined function	
`icdf_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.

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

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

  If `icdf[i]` is a string, the distribution is matched with its corresponding inverse cdf (`inv_cdf`) in the `Distributions` module (see Sec. 5.9.1) or the inverse cdf defined by 'custom_dist.py' (again see Sec. 5.9.1).

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

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

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

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

- `icdf_params`:
  Specifies the parameters for each inverse cdf in `icdf`.

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

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

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

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

  `sts_design` specifies a stratification that breaks every dimension equally into a specified number of strata of the same size. For more complex strata geometries, the strata boundaries can be explicitly defined through a text input file. See `input_file` and the corresponding documentation in Section 5.1.4.

  `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 5.1.4

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.

5.1.4 `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 5.1.3.

  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.

5.1.5 `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 5.9.1 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.

5.1.6 Adding a sampling method in `UQpy`

5.2 `Inference` Module

Coming soon...

5.3 `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

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

```
from UQpy.SampleMethods import SubsetSimulation
```

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

5.3.1 `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 5.1.5

- `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 5.1.5

- 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 5.1.5

- 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 5.1.5

- 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 5.1.5

- 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 5.8.

- model_script
  This is used to evaluate the model at each sample point using the RunModel class. For details, the user is referred to documentation for UQpy.RunModel in Section 5.8.

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

- input_script
  This is used to evaluate the model at each sample point using the RunModel class. For details, the user is referred to documentation for UQpy.RunModel in Section 5.8.

- output_script
  This is used to evaluate the model at each sample point using the RunModel class. For details, the user is referred to documentation for UQpy.RunModel in Section 5.8.

Output Attributes:

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

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

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

  By convention, failure of a given sample `sample[i][j]` is defined by `g[i][j]` < 0, where `i` indexes the conditional level and `j` indexes the sample number. For use with `SubsetSimulation`, the user's computational model must return a scalar value that follows this convention. The value is passed from `RunModel` into `SubsetSimulation` through the attribute `RunModel.model_eval.QOI` as detailed in Section 5.8.

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

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

- `pf`
  Probability of failure estimate from subset simulation

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

- MCMC_Example1.ipynb:
  In this example, the three MCMC algorithms are used to generate 1000

samples from a two-dimensional Rosenbrock pdf. The Rosenbrock pdf is defined as a function directly in the script.

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

5.3.2 `UQpy.Reliability.FORM`

Coming soon...

5.3.3 `UQpy.Reliability.SORM`

Coming soon...

5.4 `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

5.4.1 `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	$\star$	
cdf_target	Input	$\star$	
cdf_target_params	Input	$\star$	
properties	Input		$\star$
moments	Input	$\star$	
correlation	Input		$\star$
weights_error	Input		$\star$
weights_distribution	Input		$\star$
weights_moments	Input		$\star$
weights_correlation	Input		$\star$
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. 5.9.1) or the cdf defined by 'custom_dist.py' (again see Sec. 5.9.1).

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`.

5.5 `Sensitivity` Module

Coming soon...

5.6 `Optimization` Module

Coming soon...

5.7 `StochasticProcess` Module

Coming soon…

5.8 `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		⋆
`samples`	Input		⋆
`model_type`	Input		⋆
`model_script`	Input	⋆	
`input_script`	Input		⋆
`output_script`	Input		⋆
`cpu`	Input		⋆
`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 5.8.3.

- `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 5.8.1.

  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 5.8.2) 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 5.8.2) 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 5.8.2.

- `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 5.8.2.

- `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 5.8.1.

  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 5.8.2.

`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.

5.8.1 `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.

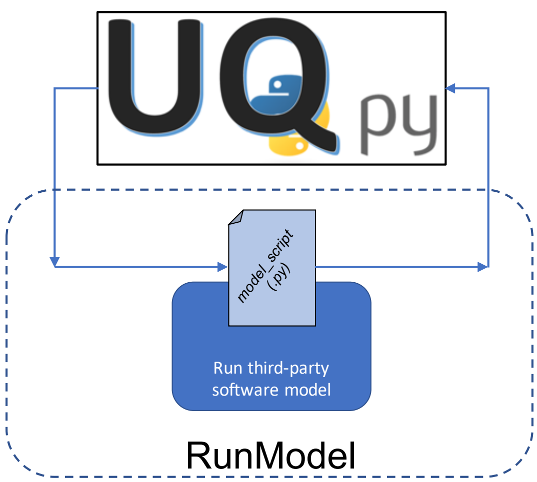

Figure 2: General workflow for running a model from a python script (`model_type` = 'pthon') using the `RunModel` class of `UQpy`.

`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:

```python
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	$\star$	
`samples`	Input	$\star$	
`QOI`	Output	$\star$	

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.

5.8.2 `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.

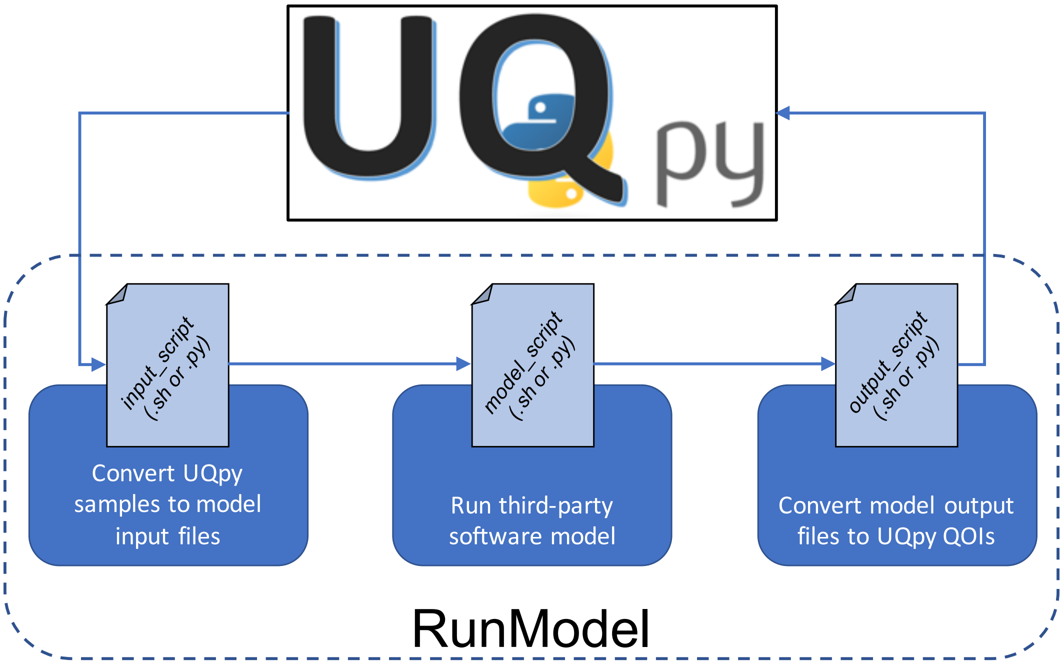

Figure 3: General workflow for running a third-party model with `UQpy` with samples and solutions passed through text files (`model_type = None`).

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 5.8.3. 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.

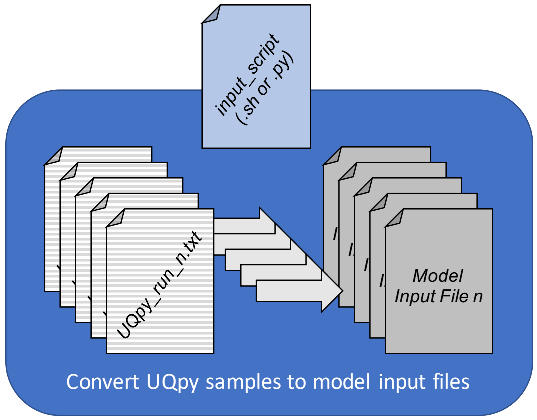

Figure 4: The user-defined `input_script` is used to read `UQpy` samples from text files defined as 'UQpy_run_n.txt' and create model input files.

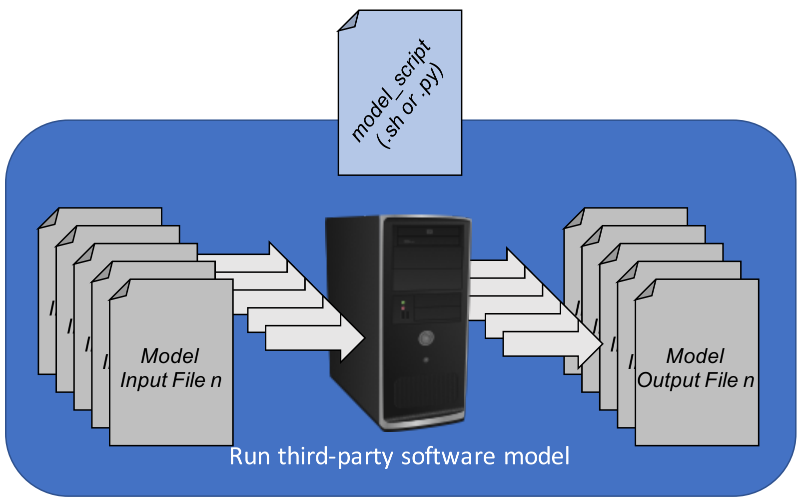

Figure 5: The user-defined `model_script` is used to run a third party software model using the model input files generated by the `input_script`. `UQpy` runs the model in a loop to evaluate all samples.

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 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 Section 5.8.3.

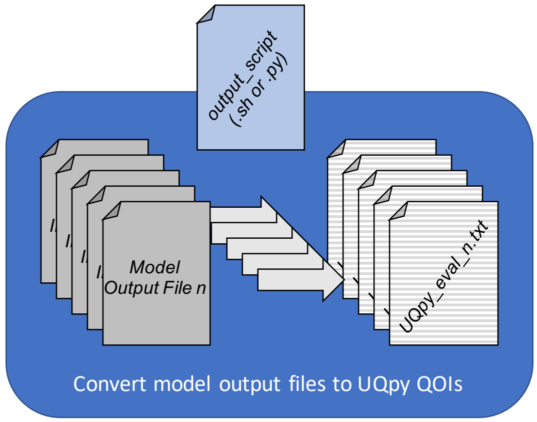

Figure 6: The user-defined `output_script` is used to post-process model results, extract a quantity of interest, and write that quantity of interest to 'UQpy_eval_n.txt' which can be read by `UQpy`.

`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 $\lfloor N/\texttt{cpu} \rfloor + \bmod\{N/\texttt{cpu}\}$ calculations on the first $\bmod\{N/\texttt{cpu}\}$ 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.

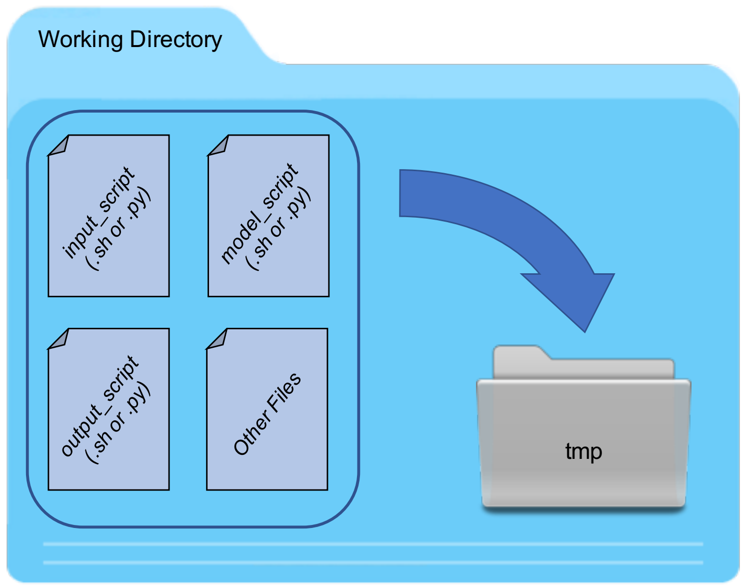

Figure 7: The first step in executing RunModel is to copy all files into a temporary subdirectory of the working directory called ‘tmp’ where all computations will be performed.

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 =

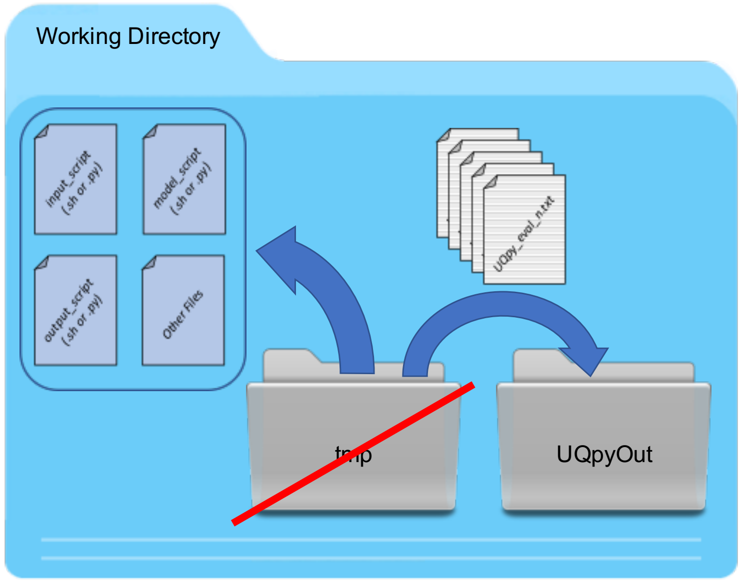

Figure 8: Final cleanup of the working director is the last step of model evaluation using `RunModel`. In the process, the input files are returned to the original working directory, all output files 'UQpy_eval_n.txt' are moved to a directory 'UQpyOut', and the 'tmp' directory is removed.

`None`' are provided that run a simple Matlab model from two-dimensional input in the following Jupyter scripts.

- 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 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.

5.8.3 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 5.8.1.

- `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.

5.8.4 Template scripts for common software applications

- Matlab
  Coming soon...

- Abaqus
  Coming soon...

- OpenSEAS
  Coming soon...

- OpenFOAM
  Coming soon...

- FEAP
  Coming soon...

- SAFIR
  Coming soon...

5.9 Supporting Modules, Functions, and Files

5.9.1 `Distributions` Module

The `Distributions` module is a support module that performs probability distribution related operations. This includes functions for computing probability densities, cumulative distributions, and their inverses 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 the following functions:

Function	**Operation**
`pdf`	Probability Density Function
`cdf`	Cumulative Distribution Function
`inv_cdf`	Inverse of Cumulative Distribution Function

All the functions of this module have the following input-output characteristics

Function I/O			
Attribute	Input/Output	**Type**	**Required**
`dist`	Input	*string*	*
`pdf`	Output	*function*	
`cdf`	Output	*function*	
`inv_cdf`	Output	*function*	

The following distributions can be accessed in `Distributions`:

- 'Uniform'
- 'Gaussian'
- 'Normal'
- 'Lognormal'
- 'Weibull'
- 'Beta'
- 'Exponential'
- 'Gamma'

Other distributions can be easily added or can be called by defining the function in `custom_dist.py`. The usage of parameters for the built-in distributions is similar to as in scipy.stats usage.

Users can also define custom distributions in 'custom_dist' file or as a function in the Python environment. The input contains a list of strings (for scipy-based distributions and user-defined distributions in 'custom_dist' file) and functions(for user-defined function within the environment). Care must be taken to define the pdf, cdf and inverse cdf of the custom distributions separately.

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 a function that computes the pdf, cdf and inverse cdf at a specified sample point for the distribution. The function definition follows standard Python scripting conventions. For compatibility with `UQpy`, each function must be defined as follows:

```
def func_name(x, params)
    value = [User-defined operations]
    return value
```

The name of the function, `func_name`, can be specified arbitrarily by the user but must be identical to the name provided as a *string* in the list `dist` described above.

The function is required to take two inputs:

- `x`: (type = *float*)
  The sample value at which to evaluate the property of the distribution.
- `params`: (type = *list*)
  A list of parameters for the probability distribution function. If the function does not require any parameters, the function must still take `params` as input. The user may then pass an empty list.

The function returns only the value of the property evaluated at `x`, defined by `value`.

An example 'custom_dist.py' file is provided with the second example from the `MCMC` class, MCMC_Example2.ipynb. See Examples from Section 5.1.5.

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.