1. UQpy Modules, Classes, & Functions

UQpy is structured in five core modules, each centered around specific functionalities:

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 simulation, or they may be deterministically drawn as in stochastic collocation or quasi-Monte Carlo.
2. Inference: This module contains a set of classes and functions to conduct probabilistic inference. The module contains methods that are based on Bayesian, frequentist, likelihood, and information theories.
3. Reliability: This module contains a set of classes and functions designed specifically to estimate probability of failure.
4. Surrogate: This module contains a set of classes and functions for building surrogate models, meta-models, or emulators.
5. Sensitivity: This module contains a set of classes and functions for performing global and local sensitivity analysis.
6. RunModel: This module contains a set of classes and functions that allows UQpy to initiate simulations using either python or third-party computational solvers.

The following sections detail the classes and functions in each module with reference to examples that illustrate their use. Guidance is based on usage in IDE Model (see Section ??)

1.1. 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
`PSS`	Partially Stratified Sampling
`MCMC`	Markov Chain Monte Carlo
`SROM`	Stochastic Reduced Order Model

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.

1.1.1. `UQpy.SampleMethods.MCS`

1.1.2. `UQpy.SampleMethods.LHS`

Property	Type	Options
`#criterion`	*string*	'random', 'centered', 'maximin', 'correlate'
`#distance`	*string*	'braycurtis', 'canberra', 'chebyshev', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'cityblock', 'matching', 'minkowski', 'rogerstanimoto', 'correlation', 'sokalmichener', 'sokalsneath', 'sqeuclidean', ' 'kulsinski', 'mahalanobis', 'russellrao', 'seuclidean',

1.1.3. `UQpy.SampleMethods.STS`

1.1.4. `UQpy.SampleMethods.PSS`

1.1.5. `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*		`algorithm` = 'MMH' or 'MH' [1,1,. . . ,1] `len` = `dimension` `algorithm`='Stretch' 2
`pdf_target_type`	*string*	'marginal_pdf' 'joint_pdf'	'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`	*nparray*
	nparray list
			array(0,0,. . . ,0)
			size = 1 × `dimension`

`nburn`	*integer*		0
`samples`	*nparray*		

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 [? ? ?].

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

- `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 [**?**].

  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 **??** 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. [**?**]. 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`$\times$`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.

1.1.6. `UQpy.SampleMethods.SROM`

1.1.7. Adding a sampling method in `UQpy`

1.2. `Inference` Module

Coming soon...

1.3. `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
`FORM`	First Order Reliability Method
`SORM`	Second Order Reliability Method

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.

1.3.1. `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		$\star$
`pdf_target_type`	Input		$\star$
`pdf_target`	Input	$\star$	
`pdf_target_params`	Input		$\star$
`pdf_proposal_type`	Input		$\star$
`pdf_proposal_scale`	Input		$\star$
`nsamples_ss`	Input	$\star$	
`algorithm`	Input		$\star$
`model_type`	Input	$\star$	
`model_script`	Input	$\star$	
`input_script`	Input	$\star$	
`output_script`	Input	$\star$	
`p_cond`	Input	$\star$	
`ss_jump`	Input		$\star$
`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
`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*		[1,1,. . . ,1] len = `dimension`
`nsamples_ss`	*integer*		`None`
`algorithm`	*string*	'MH' 'MMH' 'Stretch'	'MMH'
`model_type`	*string*	'MH' 'MMH' 'Stretch'	'MMH'
`model_script`	*string*	'MH' 'MMH' 'Stretch'	'MMH'
`input_script`	*string*	'MH' 'MMH' 'Stretch'	'MMH'
`output_script`	*string*	'MH' 'MMH' 'Stretch'	'MMH'
`p_cond`	*string*	'MH' 'MMH' 'Stretch'	'MMH'
`ss_jump`	*integer*		1
`samples`	*nparray*		

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

`g_level`	*integer*		0

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

Detailed Description of `SubsetSimulation` Class Attributes:

Input Attributes:

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

- `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 **??**

- `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 **??**

- `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 **??**

- `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 **??**

- `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 **??**

- `nsamples_ss`
  Specifies the number of samples to be generated (not including skipped states of the chain). `nsamples` must be specified. There is no default value.

- `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 [**? ? ?**].
  - 'MMH':
    Component-wise modified Metropolis-Hastings algorithm. For a description of the algorithm, see [**?**].
  - 'Stretch':
    Affine invariant ensemble sampler employing "stretch" moves. For a description of the algorithm, see [**?**].
- `model_type`
  Specifies the number of samples to be generated (not including skipped states of the chain). `nsamples` must be specified. There is no default value.
- `model_script`
  Specifies the number of samples to be generated (not including skipped states of the chain). `nsamples` must be specified. There is no default value.
- `input_script`
  Specifies the number of samples to be generated (not including skipped states of the chain). `nsamples` must be specified. There is no default value.
- `output_script`
  Specifies the number of samples to be generated (not including skipped states of the chain). `nsamples` must be specified. There is no default value.
- `p_cond`
  Specifies the number of samples to be generated (not including skipped states of the chain). `nsamples` must be specified. There is no default value.

- `ss_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.

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`$\times$`dimension`.

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

- `g_level`
  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'

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

`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.

1.3.2. `UQpy.Reliability.FORM`

1.3.3. `UQpy.Reliability.SORM`

1.4. `Surrogate` Module

Coming soon...

1.5. `Sensitivity` Module

Coming soon...

1.6. `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
```

There are two general workflows for the `RunModel` class. In the first, a model is defined or called through python scripts, which allows all message passing to be performed internally and therefore has less computational "overhead."

In the second workflow, information is passed between UQpy and a third-party solver through text files. The following sections detail these two workflows.

1.6.1. 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 1.2.3. Figure 1 shows a general flow-chart for the RunModel class.

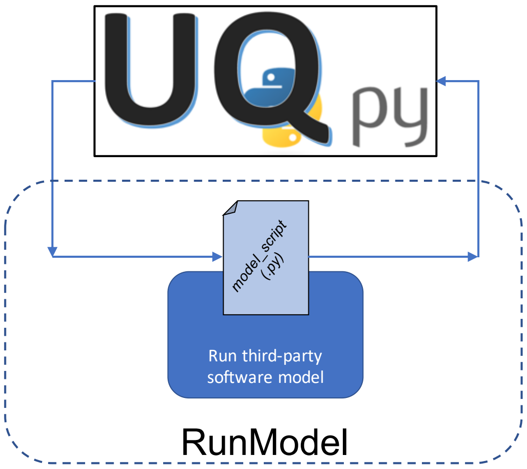

Figure 1: General workflow for running a model from a python script (model_type = 'pthon') using the RunModel class of UQpy.

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

```
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 call a third-party model (e.g. Matlab, Abaqus, or a custom simulation code).

`RunPythonModel` C		
Attribute	**Type**	**Description**
`dimension`	*integer*	Dimension of the samples array.
`samples`	*nparray*	Sample points at which to evaluate the model.
`QOI`	*nparray*	A list containing the quantity of interest returned from the model. Each i

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 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.

1.6.2. `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. 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.

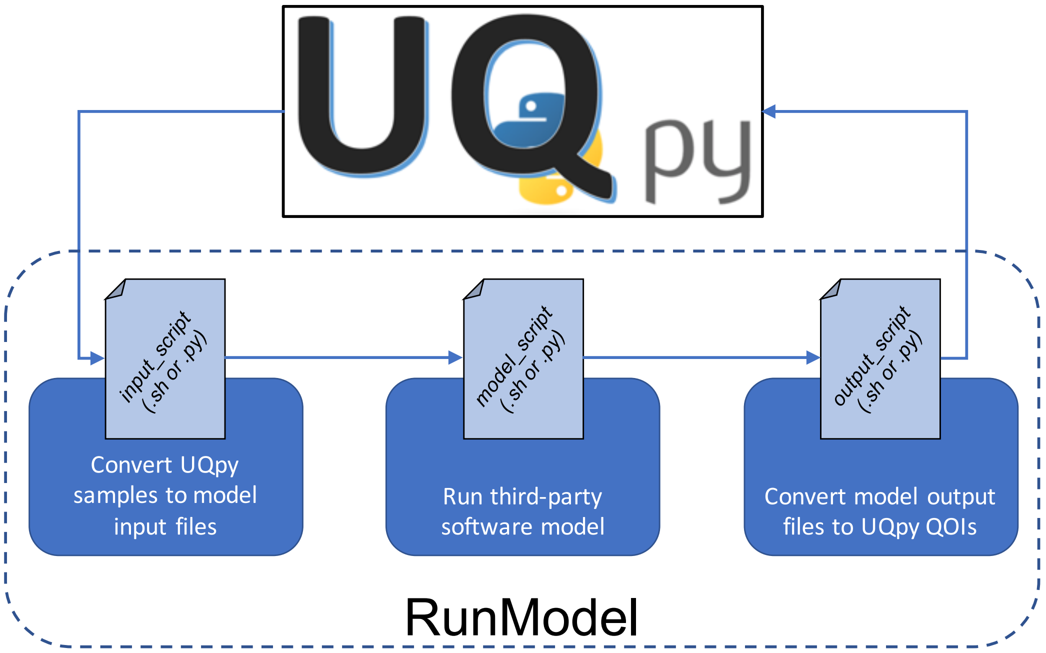
UQ py
input_script (.sh or .py)
model_script (.sh or .py)
output_script (.sh or .py)
Convert UQpy samples to model input files
Run third-party software model
Convert model output files to UQpy QOIs
RunModel

Figure 2:

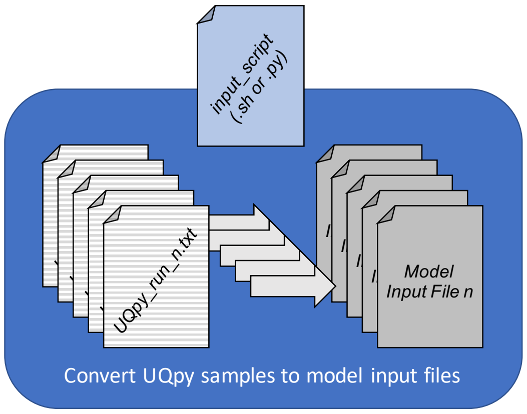

Figure 3:

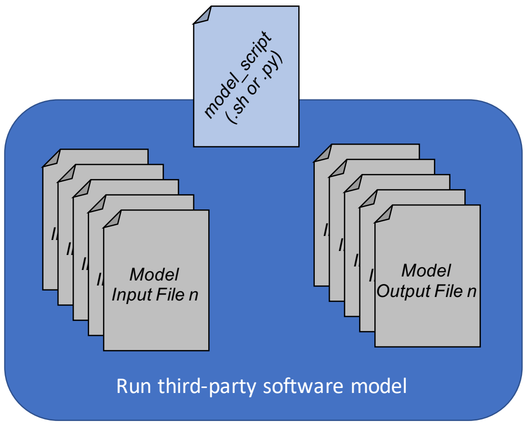

Figure 4: Need to edit this to include the model running on a computer

1.6.3. Calling `RunModel` and defining its attributes

1.6.4. Necessary files and scripts

1.7. Supporting Modules, Functions, and Files

1.7.1. `Distributions` Module

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

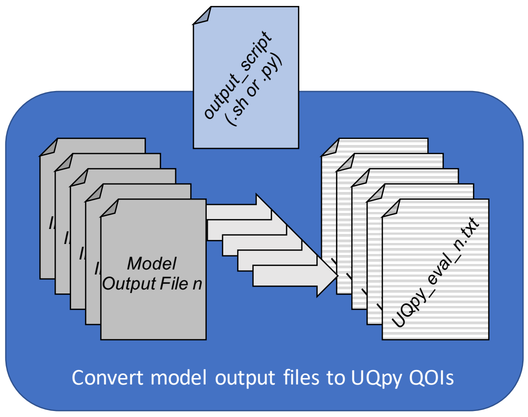
output_script (.sh or .py)
Model Output File n
UQpy_eval_n.txt
Convert model output files to UQpy QOIs

Figure 5:

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

The input and output of the `pdf` function are described in the table below.

`pdf` Function I/O			
Attribute	Input/Output	**Type**	**Options**
`dist`	Input	*string*	Custom
`return`	Output	*float*	N/A

The `pdf` function enables the evaluation of a standard pdf or an arbitrary user-defined probability density function. When a custom pdf is used, the pdf is defined through the Python script 'custom_pdf.py', which must be located in the current working directory. Details follow.

Description of `custom_pdf.py`

The script 'custom_pdf.py' allows the user to define a custom probability density function. In the script, the user may define a function that computes the pdf at a specified sample point. The function definition follows standard Python scripting conventions. For compatibility with `UQpy`, each function must be defined as follows:

```
def func_name(x, params)
    pdf_value = [User-defined operations]
    return pdf_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* to the value of `dist` from the `pdf` function described above.

The function is required to take two inputs:

- `x`: (type = *float*)
  The sample value at which to evaluate the probability density function.
- `params`: (type = *list*)
  A list of parameters for the probability density 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 pdf evaluate at `x`, defined by `pdf_value`.

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