UQpy - Uncertainty Quantification with Python

Dimitris G. Giovanis, M. D. Shields

Johns Hopkins University, USA

1. Installing UQpy

Prerequisites: You have at least one Python interpreter 3.6+ properly installed on your computer. In order to get the latest experimental version of UQpy the code can be installed from Github directly as follows:

```
$git clone https://github.com/SURGroup/UQpy.git
$cd UQpy/
$pip install -r requirements.txt.
$python setup.py install.
```

The last command might need **sudo** prefix, depending on your python setup.

2. Overview

UQpy (Uncertainty Quantification (UQ) using python) is a software toolbox containing a collection of modules written in Python that provide standardized solutions for many UQ problems that occur in physical model. Connection between UQpy and the user-defined computational model is made with text-based and bash shell script(s) provided by the user. Execution of UQpy results in realizations of the parametric space of interest using advanced techniques, as well as evaluations the corresponding model responses. UQpy is entirely code-agnostic and gives users a fully functional tool for performing UQ with nearly any computational analysis code. UQpy performs submission, execution, monitoring and post-process analysis, specifically tailored to the

analysis tool and the available platform and thus, it is amenable to performing adaptive UQ methods. `UQpy` is written in the Python 3 programming language.

2.1. Compiled version of `UQpy`

We need to address the Windows version

2.2. Interpreted version of `UQpy`

The interpreted version of `UQpy` requires a Python shell supporting Python 3.6+ as well as several common Python libraries as well. After downloading and installing `UQpy`, the following `UQpy`-specific files are required and must be co-located in the subdirectory lib/**UQpy**, which is in the same directory as UQpy_cmd.py:

- UQpyModules.py - Contains various functions.
- SampleMethods.py - Contains the available sampling methods used for exploring the parameter space.
- ReadInputFile.py - Reads the necessary UQ Parameter data file in case of running `UQpy` via command line, and converts it to python variables.
- PDFs.py - Contains the percent point functions of all the supported distributions; any new distribution can be added here.

3. Using UQpy- Required files

`UQpy` may be run using either an Integrated Development Environment (IDE) used in computer programming, specifically for the Python language or via the command line. The interpreted version of `UQpy`, has been tested to run in IDE PyCharm 2017.3.3.

In order to use `UQpy` for evaluating the response of any computational model for a number of parameter realizations, `UQpy` requires three **executable**[1] bash shell scripts:

[1] `$chmod +x` name1*.sh

- name1*.sh for linking the analysis software to `UQpy`
- name2*.sh for converting the the file containing the parameter values (text-based file) into appropriate input file for the analysis code
- name3*.sh converting the result of the software analysis into an appropriate (text-based) file to be read from `UQpy`. This is necessary in case of running adaptive UQ methods and/or post-processing of the results.

The names of these files are user defined. Additional to these files, if the user wants to generate the realizations of the random parameters according to one of the available sampling methods provided in `UQpy`, it is necessary to provide an text-based file under the name (UQpy_params.txt), which will enclose all the probabilistic information required for running the selected sampling method. T

The aforementioned files are directly specified by the user and may be in any directory.

4. UQpy Usage

`UQpy` is user friendly since it only requires the user to have basic knowledge in writing bash shell scripts.

4.1. Using the `UQpy` Command Line Mode

`UQpy` can be executed directly through the command line. It is provided as an option to the user who doesn't have sufficient familiarity and experience with python. Command line execution is advantageous when analyses need to be performed on a high-performance computing systems without direct graphics capability. In order to execute the interpreted version `UQpy` from the command line the user needs to change to the `UQpy` directory and then type in terminal :

```
$python UQpy_cmd.py --dir pathToModel --model name1*.sh --input name2*.sh --output name3*.sh
```

Where

- UQpy_cmd.py is the python script that actually runs `UQpy` via command line and needs to be located in the directory **UQpy**.

- `--dir` is the absolute path to the folder which contains the necessary files {name1*.sh , name2*.sh , name3*.sh and UQpy_params.txt}.
- `--model` points to the name1*.sh bash script
- `--input` points to the name2*.sh bash script
- `--output` points to the name3*.sh bash script

In order for `UQpy` to run from command line the file UQpy_params.txt is necessary to be located inside `--dir` otherwise, the execution will return an error. However the user may skip the entries {`--input, --output, --model`} if `UQpy` is utilized only for generating realizations of the random parameter and not for model evaluations. Another optional entry for the user is `--CPUs` which sets the number of processors used for the evaluation of the model, in case of parallel processing. The user can see all the available options (Fig. 1) by typing in terminal

```
$python UQpy_cmd.py --help
```

which results in:

```
python UQpy.py --{options}

optional arguments:
  -h, --help            show this help message and exit
  --dir MODEL_DIRECTORY
                        Specify the location of the model's directory.
  --input INPUT_SHELL_SCRIPT
                        Specify the name of the shell script *.sh used to
                        transform the output of UQpy (UQpyOut_*.txt file) into
                        the appropriate model input file
  --output OUTPUT_SHELL_SCRIPT
                        Specify the name of the shell script *.sh used to
                        transform the output of the model into the appropriate
                        UQpy input file (UQpyInp_*.txt)
  --model SOLVER        Specify the name of the shell script used for running
                        the model
  --CPUs CPUS           Number of local cores to be used for the analysis
```

Figure 1:

4.2. Using the UQpy IDE Mode

After installation, UQpy is build in the local Pythons standard library and thus, it runs from any Integrated Development Environment (PyCharm, Atom, Eclipse, e.t.c) which provides code analysis and debugging. In order to use UQpy libraries in a project the user needs to import the specific module to its workspace. This can be done by writing in a python script

```
from UQpy import *
```

which will load all modules of UQpy. If a specific class from the sample methods (e.g Monte Carlo simulation) is required then the user can selectively load it to the project by typing

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

This functionality of UQpy enables the independent usage of its modules, which makes UQpy a powerful tool for UQ analysis and communication between python and various computational codes of different nature. In order to generate 100 realizations of two random parameters using MCS the user needs to type:

```
from UQpy.SampleMethods import MCS
x = MCS(dimension=2, pdf_type=['Uniform', 'Uniform'])
     pdf_params=[[0, 1], [0, 1] ], nsamples=100)
```

This will create the object x with is properties:

1. pdf_type: type of distribution for each parameter
2. pdf_params: distribution parameters
3. nsamples: number of samples to be generated
4. dimension: number of random parameters
5. samples: generated samples in the parameter space
6. samplesU01: generated samples in the Uniform space, $U[0, 1]^{\texttt{dimension}}$

5. UQpy workflow

6. Templates for the required Files

The interaction between UQpy and any external solver is made with text-based files which are simple to process and easy to work with in python.

6.1. Probabilistic Parameter File

The file that keeps the probabilistic properties of the parameters should always be under the name:

UQpy_params.txt

Creating UQpy_params.txt is simple and straightforward; Each property that is required for the selected sampling method, is defined in a line that starts with a hash-tag (#), followed by a key-word and/or key-phrase (case sensitive) describing the property[2]. The file ends with the key-word `#end`. Under that line, the specific attributes of the property are defined, according to the UQpy available options. Thus, different sampling methods require different parameter file .

6.1.1. Required properties for various sampling methods

The properties that need to be specified by the user inside the parameter file in order to run different sampling methods, for exploring the parameter space. A summary of these properties is given next:

[2]The order that the properties are declared in UQpy_params.txt is not important .

Monte Carlo simulation		
Property	**Mandatory**	**Optional**
`#method`	$\star$	
`#number of samples`	$\star$	
`#number of parameters`	$\star$	
`#distribution type`	$\star$	
`#distribution parameters`	$\star$	
`#names of parameters`		$\star$
`#SROM`		**True** or **False**

Latin hypercube simulation		
Property	**Mandatory**	**Optional**
`#method`	$\star$	
`#number of samples`	$\star$	
`#number of parameters`	$\star$	
`#distribution type`	$\star$	
`#distribution parameters`	$\star$	
`#names of parameters`		$\star$
`#criterion`		$\star$
`#distance`		$\star$
`#metric`		$\star$
`#SROM`		**True** or **False**

Stratified sampling		
Property	**Mandatory**	**Optional**
`#method`	$\star$	
`#distribution type`	$\star$	
`#number of parameters`		$\star$
`#distribution parameters`	$\star$	
`#design`	$\star$	
`#names of parameters`		$\star$
`#SROM`		**True** or **False**

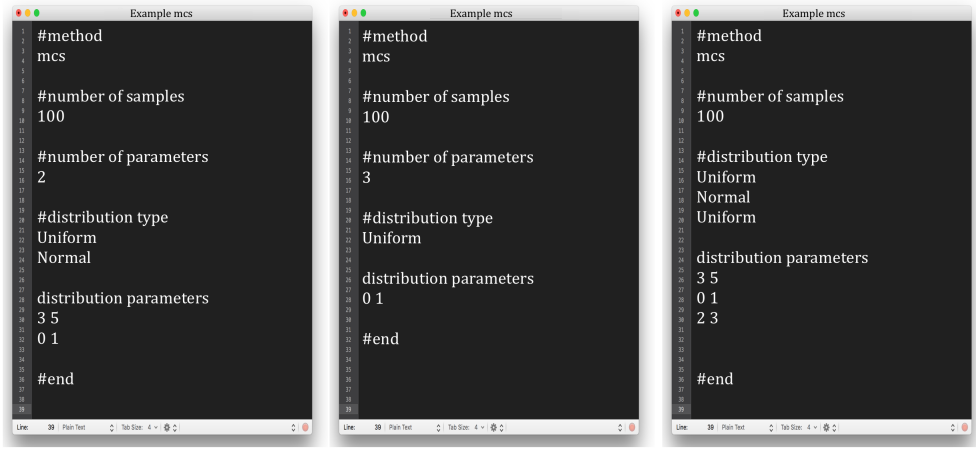
Partially Stratified sampling		
Property	**Mandatory**	**Optional**
`#method`	$\star$	
`#distribution type`	$\star$	
`#distribution parameters`	$\star$	
`#number of parameters`		$\star$
`#design`	$\star$	
`#strata`	$\star$	
`#names of parameters`		$\star$
`#SROM`		**True** or **False**

Stochastic reduced order model		
Property	**Mandatory**	**Optional**
If `#SROM` property is **True**		
`#moments`	$\star$	
`#error function weights`	$\star$	
`#properties to match`		$\star$
`#sample weights`		$\star$

6.1.2. Examples of parameter files

Special instruction on how to create the parameter file that will enclose the required properties of the selected sampling method are the following :

- A complete parameter file for e.g. Monte Carlo simulation can defined like Fig.2(a).
- If all random parameters follow the same distribution type with the same distribution parameters then a parameter file can defined like Fig.2(b) where the distribution type and parameters need to be defined once. In this case existence of the property "number of random parameters" is mandatory.
- For the case the number of distribution type is equal to the number of distribution parameters (Fig.2(c)) then, definition of property "number of parameters" is optional .



(a) (b) (c)

Figure 2:

6.2. Template Input File

The functionality of the name2*.sh bash shell script file is to convert the text-based output file of `UQpy` (UQpy_run_i.txt) that contains the realization i of the parameter vector into appropriate input for the analysis code. The user is responsible for creating the appropriate bash script for performing this action. For example, if the software code reads a text-based file called modelInput_i.txt then a possible name2*.sh script would be the one depicted in Fig.3; it is used for renaming UQpy_run_i.txt to modelInput_i.txt.

6.3. Template Model File

In order for `UQpy` to execute the software code a bash script (name1*.sh) is necessary.

6.4. Template Output File

The functionality of the name3*.sh bash shell script file is to convert the output of the code analysis (which can be at any format) into a text file file under the name UQpy_eval_i.txt", where i refers to the number of simulation, ready to be processed by `UQpy`. This step is required for running adaptive

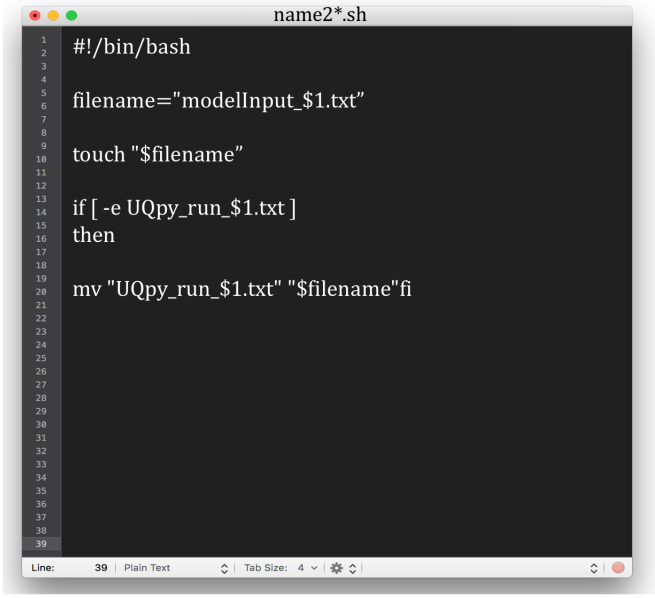


Figure 3:

UQ methods as well as for post-processing of the result but in any case it is mandatory to provide such file. For example, if the software code generates a text-based file called solution_i.txt then a possible name3*.sh script would be the one depicted in Fig.5; it is used for renaming solution_i.txt to UQpy_eval_i.txt".

7. `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.

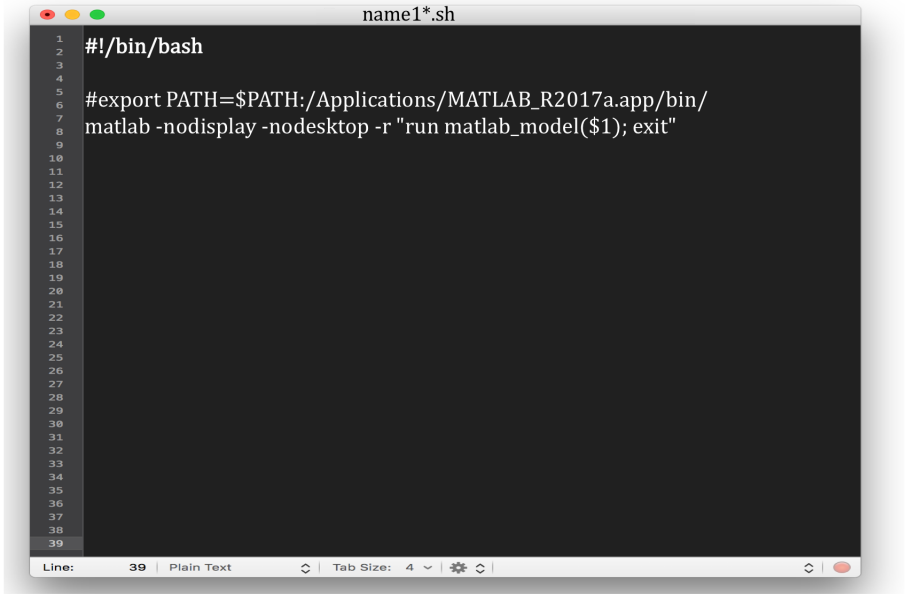


Figure 4:

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 4.2)

7.1. `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:

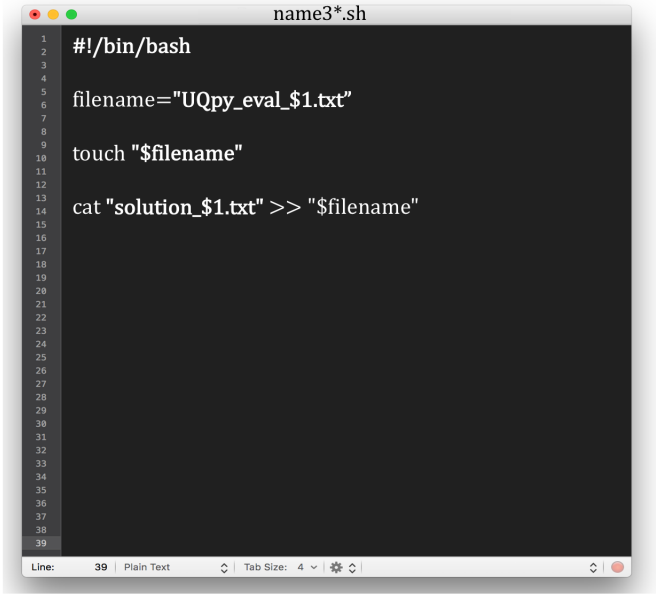


Figure 5:

```
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.

7.1.1. `UQpy.SampleMethods.MCS`

7.1.2. `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',

7.1.3. `UQpy.SampleMethods.STS`

7.1.4. `UQpy.SampleMethods.PSS`

7.1.5. `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		$\star$
`pdf_proposal_type`	Input		$\star$
`pdf_proposal_scale`	Input		$\star$
`pdf_target_type`	Input		$\star$
`pdf_target`	Input	$\star$	
`pdf_target_params`	Input		$\star$
`algorithm`	Input		$\star$
`jump`	Input		$\star$
`nsamples`	Input	$\star$	
`seed`	Input		$\star$
`nburn`	Input		$\star$
`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] `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 $\times$ `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 [1, 2, 3].
  - 'MMH':
    Component-wise modified Metropolis-Hastings algorithm. For a description of the algorithm, see [3].
  - 'Stretch':
    Affine invariant ensemble sampler employing "stretch" moves. For a description of the algorithm, see [4].

- `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 [4].

  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 7.7.1 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. [4]. 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.

7.1.6. `UQpy.SampleMethods.SROM`

7.1.7. Adding a sampling method in `UQpy`

7.2. `Inference` Module

Coming soon...

7.3. `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.

7.3.1. 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		⋆
`pdf_target_type`	Input		⋆
`pdf_target`	Input	⋆	
`pdf_target_params`	Input		⋆
`pdf_proposal_type`	Input		⋆
`pdf_proposal_scale`	Input		⋆
`nsamples_ss`	Input	⋆	
`algorithm`	Input		⋆
`model_type`	Input	⋆	
`model_script`	Input	⋆	
`input_script`	Input	⋆	
`output_script`	Input	⋆	
`p_cond`	Input	⋆	
`ss_jump`	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
`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
`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 × `dimension`
`g_level`	*integer*		0
`pf`	*nparray* *nparray list*		array(0,0,...,0) size = 1 × `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 7.1.5

- `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 7.1.5

- `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 7.1.5

- `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 7.1.5

- `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 7.1.5

- `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 [1, 2, 3].
- ‘MMH’:
  Component-wise modified Metropolis-Hastings algorithm. For a description of the algorithm, see [3].
- ‘Stretch’:
  Affine invariant ensemble sampler employing “stretch” moves. For a description of the algorithm, see [4].

- `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`×`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. [4]. 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. [4]. 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.

7.3.2. `UQpy.Reliability.FORM`

7.3.3. `UQpy.Reliability.SORM`

7.4. `Surrogate` Module

Coming soon...

7.5. `Sensitivity` Module

Coming soon...

7.6. `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.

7.6.1. `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 7.6.3. Figure 6 shows a general flow-chart for the `RunModel` class.

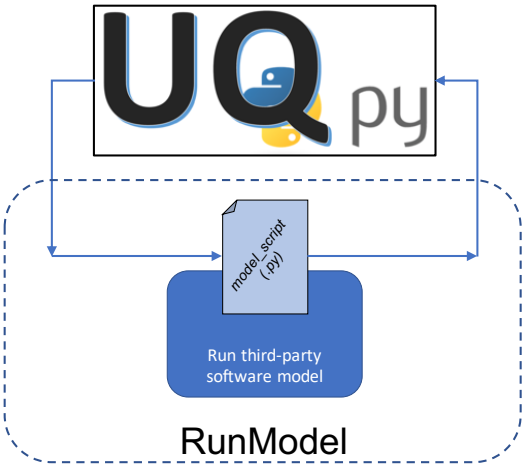


Figure 6: General workflow for running a model from a python script (`model_type` = 'pthon') using the `RunModel` class of `UQpy`.

`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.

7.6.2. `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.

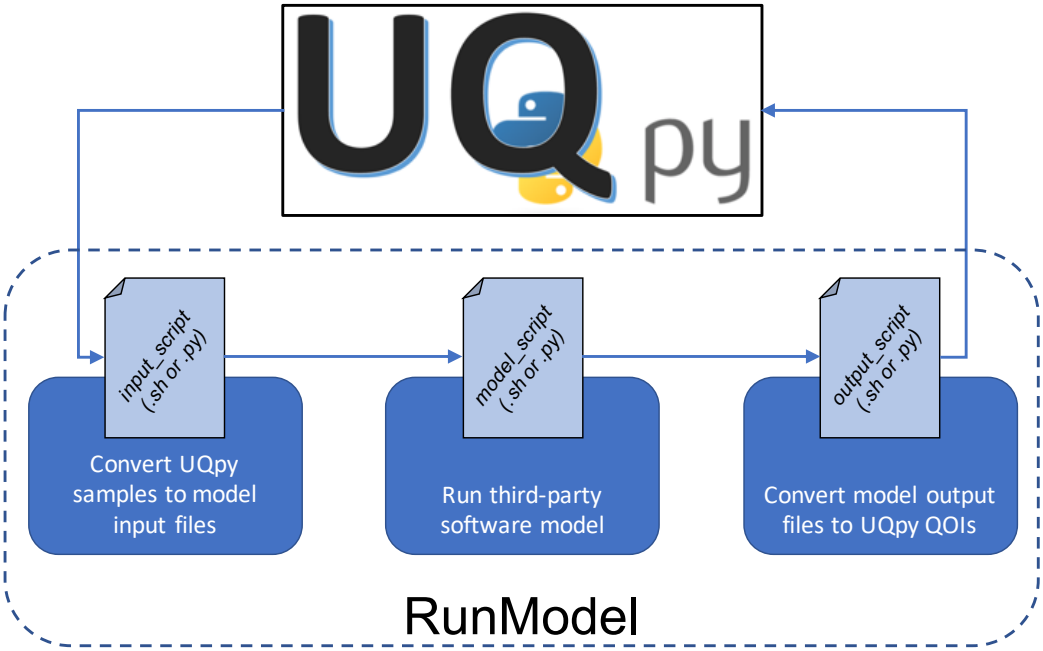


Figure 7:

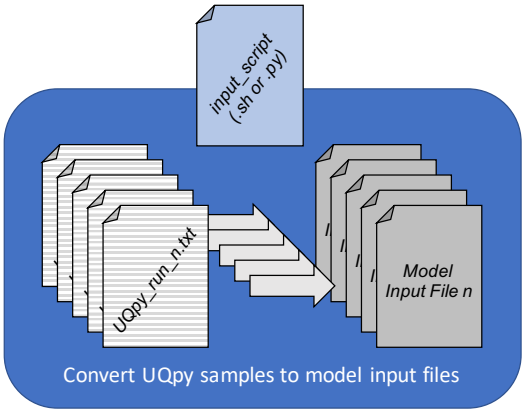


Figure 8:

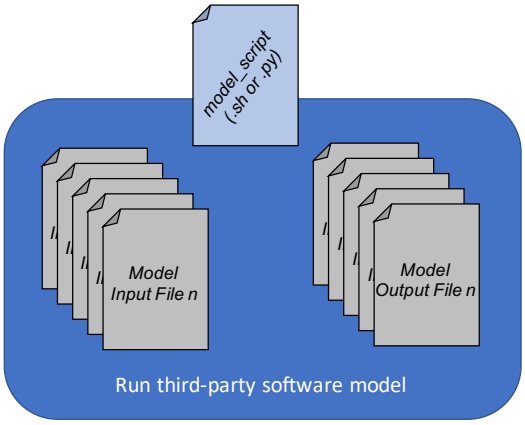


Figure 9: Need to edit this to include the model running on a computer

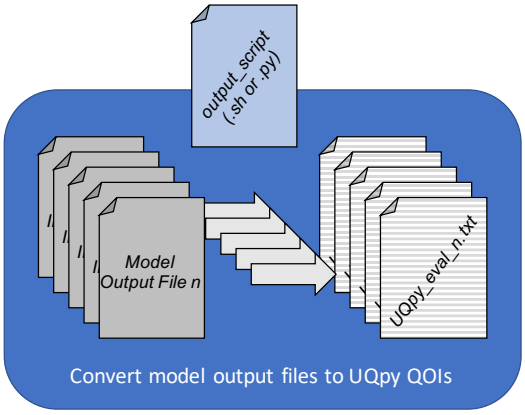


Figure 10:

7.6.3. Calling `RunModel` and defining its attributes

7.6.4. Necessary files and scripts

7.7. Supporting Modules, Functions, and Files

7.7.1. `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.

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 7.1.5.

References

[1] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, E. Teller, Equation of State Calculations by Fast Computing Machines, The Journal of Chemical Physics 21 (1953) 1087.

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

[3] S.-K. Au, J. L. Beck, Estimation of small failure probabilities in high dimensions by subset simulation, Probabilistic Engineering Mechanics 16 (2001) 263–277.

[4] J. Goodman, J. Weare, Ensemble samplers with affine invariance, Communications in applied mathematics and computational science 5 (2010) 65–80.