Journal of Computational Science 47 (2020) 101204

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

UQpy: A general purpose Python package and development environment for uncertainty quantification

Audrey Olivier, Dimitris G. Giovanis, B.S. Aakash, Mohit Chauhan, Lohit Vandanapu, Michael D. Shields *

Department of Civil and Systems Engineering, Johns Hopkins University, Baltimore, MD, United States

ARTICLE INFO

Keywords:
Uncertainty quantification
Computational modeling
High-performance computing
Python
Software

ABSTRACT

This paper presents the UQpy software toolbox, an open-source Python package for general uncertainty quantification (UQ) in mathematical and physical systems. The software serves as both a user-ready toolbox that includes many of the latest methods for UQ in computational modeling and a convenient development environment for Python programmers advancing the field of UQ. The paper presents an introduction to the software's architecture and existing capabilities, divided in the code in a set of modules centered around different UQ tasks such as sampling methods, generation of random processes and random fields, probabilistic inverse modeling, reliability analysis, surrogate modeling, and active learning. The paper also highlights the importance of the RunModel module, which is used to drive simulations in the uncertainty analyses performed in UQpy. This module conveniently allows the user to define computational models directly in Python, or to run simulations from a third-party software in serial or in parallel. To illustrate the various capabilities, two examples are tracked throughout the paper and analyzed repeatedly for various UQ tasks. The first is a Python model solving a nonlinear structural dynamics problem, used to illustrate UQpy's capabilities in sampling and forward propagation of high dimensional random vectors (stochastic processes), and probabilistic inference. The second model is a third-party Abaqus finite element model solving the thermomechanical response of a beam structure. This example is used to illustrate UQpy's capabilities in variance reduction sampling techniques, reliability analysis, surrogate modeling and active learning techniques.

1. Introduction: UQpy purpose and workflow

Uncertainty quantification (UQ) is the science of quantifying, characterizing, and reducing uncertainty in computational and real world systems. It finds applications in various fields of science and engineering, such as stochastic mechanics and structural reliability [1,2], multi-scale modeling [3], biological systems [4], climate modeling [5] or hydrology [6–9]. This paper presents UQpy, an open-source, general purpose Python package for modeling uncertainty in physical and mathematical systems. The UQpy package is available for download from Github [10], along with all Python scripts necessary to run the examples presented in the manuscript [11], which are also included as supplementary materials. The code is organized as a set of modules centered around core capabilities in UQ, each represented by a box in Fig. 1. The modules build from foundational probabilistic operations to advanced methodologies where each module contains various classes that can easily invoke one another and can be combined to perform complex UQ tasks, as will be demonstrated throughout the manuscript. This object-oriented architecture also allows the user to easily add new capabilities. This is illustrated in Fig. 1 by the addition of new modules for advanced analysis, but extension of existing modules is similarly straightforward and does not require intrusion with existing classes and functions, as discussed in more detail below.

This flexible package can be utilized in various ways, depending on the user's goals, Python coding proficiency and desired level of interaction with the code. On one-hand, UQpy can be used in a "black-box" fashion as it provides a range of user-ready algorithms that can be easily imported into the user's Python environment and invoked to perform various UQ tasks such as forward uncertainty propagation, inverse learning or estimation of failure probabilities to name only a few, with minimal interaction with the code itself. On the other hand, it is fully open-source and was designed with a conveniently extensible, object-oriented architecture. Considering this flexibility, it is useful to think of UQpy in two ways: 1. As a UQ toolbox for the casual or advanced user;

* Corresponding author.

https://doi.org/10.1016/j.jocs.2020.101204
Received 23 February 2020; Received in revised form 29 July 2020; Accepted 13 August 2020
Available online 24 August 2020

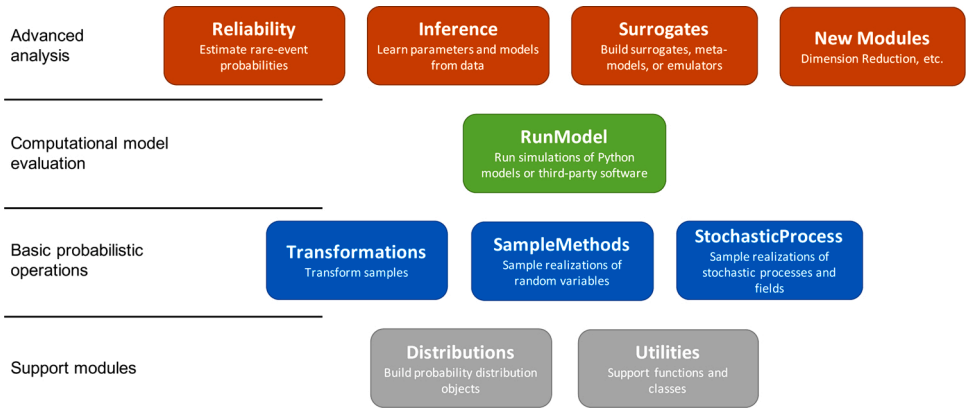


Fig. 1. UQpy modules organized from their most fundamental to those for advanced modeling. Moving upward in the figure, the modules at higher levels generally leverage those at the lower levels.

2. As a development environment through which to advance the field of UQ in computational modeling.

Before getting into the code, it is important to recognize that several codes currently exist for UQ and perhaps shed some light on what sets `UQpy` apart. The following paragraphs aim to provide a review of Python based and non-Python based toolboxes for general UQ, though recognizing that it may not provide a fully exhaustive list. In particular, it is noted that numerous Python codes exist that focus on one specific aspect of UQ, such as MCMC, and our review does not include many of these tools. Instead, we aim to review those tools that are general and incorporate a broad range of UQ functionalities.

Some codes, such as the Computational Stochastic Structural Analysis (COSSAN) software [12] began development as early as 1992. COSSAN development has continued to present day and the software has now split into two components. OpenCossan is an open-source Matlab toolbox for UQ that, like `UQpy`, is available for download via Github. COSSAN-X meanwhile comprises a set of compiled toolboxes for various tasks in UQ. Another widely used and powerful Matlab toolbox is the UQLab software [13] developed by the Chair of Risk, Reliability, and Uncertainty Quantification at ETH Zurich. The Engineering Risk Analysis Group at the Technical University of Munich has also developed a variety of MATLAB and Python tools for various UQ tasks [14]. Other Matlab toolboxes that are available are filling specific niches within UQ such as the SUrrogate MOdeling (SUMO) toolbox [15] and the Finite Element Reliability Using Matlab (FERUM) toolbox [16]. Various packages are also available for the R statistical software including DiceDesign, DiceKriging, and DiceOptim [17,18], the mistral package [19] for reliability analysis, and the sensitivity package [20].

Among the developed software, some commercial and/or industrial codes have also begun to arise. Perhaps the most widely used UQ software is the Dakota package [21] developed by Sandia National Laboratories (SNL). Dakota is an open-source C++ package with many advanced features that is widely used across the US government labs. Also developed at SNL is the C++/Python UQ Toolkit (UQTk) [22]. Additional notable UQ packages include the Open source Treatment of Uncertainty, Risk 'N Statistics (Open TURNS) C++ package [23], the NESSUS packaged developed by Southwest Research Institute [24] and the SMARTUQ package.

The primary motivation for developing `UQpy` is the lack of comprehensive UQ package specifically for the Python language, which is extensively used for scientific computing. While extensive packages are available in Matlab, C++, and R, only relatively few disparate codes were previously available in Python. These tools were either for specific niche applications, were catered to specific methodologies, or were only partially Python-based (e.g. having a Python interface). The UncertainPy code [25], for example, is catered specifically to applications in computational neuroscience. pyROM [26] is an open source Python computational framework that implements model reduction techniques. Chaospy [27], meanwhile, performs UQ using polynomial chaos expansions, which makes it an important contribution but not a general toolbox. In fact, the existence of Chaospy is one reason that polynomial chaos has not yet been implemented in `UQpy`, although it is expected to be added in the near future. A number of Python packages are tailored for inverse uncertainty quantification, such as SPUX [28] or ABCpy [29] which provide Python implementations of Bayesian calibration algorithms or SPOTpy that implements a number of algorithms for statistical parameter optimization. Packages such as the MIT Uncertainty Quantification (MUQ), Open TURNS, Korali [30] and UQTk toolboxes do include a variety of tools and algorithms for general purpose uncertainty quantification, however they are not exclusive Python packages. Instead they are either mixed Python/C++ codes or C++ codes with a Python interface, and thus are not particularly well-suited for Python development of new UQ methodologies and algorithms. Finally, the UQ-PyL package [31] is a fully Python-based software platform that includes various UQ tasks tailored to quantifying and reducing model uncertainties associated with model parameters. UQpy's scope is somewhat broader as it allows for modeling of a wider range of uncertainties via stochastic processes/fields for instance. Also, though new algorithms can be added to the UQ-PyL package by creating new python scripts, UQpy's object oriented architecture is better suited for development purposes, as will be shown throughout the manuscript. `UQpy` is therefore intended to be a fully general UQ toolbox and development environment for Python. As such, it combines many of the most widely used and advanced methodologies in an architecture that can be imported directly into the user's Python environment, can seamlessly link to any third-party computational model, is conveniently extensible for development of new methodologies, and is capable of harnessing high performance computing resources.

The following provides a brief overview of the code (specifically Version 3), and how it can be adopted by interested users to perform readily available UQ tasks or utilized by more advanced users for UQ development activities.

1.1. `UQpy` *as a toolbox*

`UQpy` has a wide-range of built-in capabilities that are ready to use. These capabilities are divided into a set of modules, each centered around a common objective, as illustrated in Fig. 1. Utilization of `UQpy` is built upon fundamental probabilistic operations and the evaluation of computational models via the `RunModel` module, which drives all simulations in the uncertainty analysis. `RunModel` allows the user to define the model directly in Python (in which case the model is imported into the user's Python environment) or to run simulations from a third-party software. Many other modules, therefore, rely on this `RunModel`

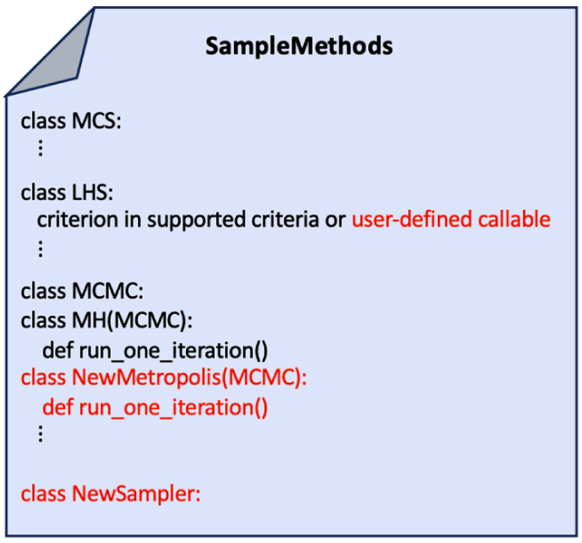


Fig. 2. Illustration of code modifications to add classes and methods to `UQpy` modules.

object to perform simulations required by the various UQ tasks. In this context, `UQpy` can be non-intrusively wrapped around any user software, enhancing it to enable consideration of uncertainties. The `RunModel` module is discussed in more detail in Section 2.

Built around the `RunModel` module are six modules for UQ functionalities ranging from fundamental operations (lower level) to advanced methodologies (upper level). Within the modules, each specific functionality is implemented as a class. For example, the `SampleMethods` module contains a set of classes designed to draw samples, randomly or deterministically, from a specified parameter space. This module includes classes ranging from simple Monte Carlo sampling (`SampleMethods.MCS`) to advanced Markov chain Monte Carlo methods (`SampleMethods.MCMC`). A complete list of modules and classes in the current release (Version 3.0) is provided in Appendix A. Note that additional modules are under development as discussed in Appendix B. Through this modular architecture, the user can easily take advantage of `UQpy` to perform a variety of UQ tasks of various complexity. The user may be interested in simple functionalities that leverage a single class of `UQpy`, e.g., sampling realizations of a stochastic process via `StochasticProcess.SRM`. More importantly though, the user can use `UQpy` to perform much more complex UQ tasks that internally leverage various `UQpy` modules. For instance, the `SampleMethods.RSS` class performs adaptive sampling of random variables by iteratively leveraging the `RunModel` and `Surrogates.Kriging` classes, while the `Reliability.SubsetSimulation` class internally leverages the `RunModel` and `SampleMethods.MCMC` classes to compute rare-event probabilities. Additionally, thanks to their modular nature, the user can "daisy-chain" various classes of the software to cater existing capabilities to their specific needs.

Those interested in using `UQpy` as a UQ toolbox have a variety of options to install the software as discussed in the software documentation [32].

1.2. `UQpy` as a development environment

Perhaps more importantly, from a scientific and research perspective, `UQpy` is specifically designed to serve as a platform for developing new UQ methodologies and algorithms. Firstly, contrary to packages that are not fully Python-based, `UQpy` can very easily inter-operate with other Python code, from widely used packages such as `Numpy` and `Scipy` for mathematical operations or `Matplotlib` for plotting, to more specialized packages for modeling (e.g., `SfePy` [33] for finite element modeling, `scikit-learn` for machine learning and surrogate modeling). Furthermore, its modularized, object-oriented architecture is designed such that extensions or modifications to the code can easily be performed to tackle a wider range of problems. Such extensions can be implemented in various ways, either through development of new modules (see Fig. 1) or through the extension of existing modules with new classes and/or functions – as illustrated conceptually in Fig. 2 for extensions to the `SampleMethods` module.

The code is designed such that its extension requires minimal intrusion with existing classes and functions. In the simplest case, new capabilities are implemented by simply adding a new class to an existing module as illustrated by the `NewSampler` class in Fig. 2. Any such new class can leverage the full suite of existing capabilities in `UQpy`. Where appropriate, the code relies on inheritance concepts that greatly facilitate development of new methods. In such cases, the parent class controls the framework and defines generic attributes and methods that are shared across all sub-classes. Specific algorithms are created by creating child classes that implement new methods or over-write only those methods of the parent class that the user wishes to change. In this fashion, a developer who wishes to add a new algorithm need not modify any of the existing code, only add a class that inherits from the parent class, as illustrated by the `NewMetropolis` sub-class of the parent `MCMC` class in Fig. 2. This specific case is elaborated in greater detail in Section 3.1.2. In other instances, the code allows replacing supported functionalities with custom functions. An example in the `SampleMethods` module is the `LHS` class, where four criteria are currently supported for pairing the samples. However a user can also provide a custom function that pairs the samples, thus non-intrusively enhancing the existing code in a straightforward manner.

Because a primary objective of this work is to illustrate how `UQpy` serves as a development environment for UQ research, additional details will be provided throughout the text below to specifically illustrate how new developments can be made in the various modules. Those interested in developing with `UQpy` are encouraged to install the software from Github using a developer install as described in the software documentation [32].

1.3. Structure of the paper

The structure of the paper follows the structure of the software. `UQpy` currently has nine modules, seven primary modules for probabilistic modeling and two support modules, as shown in Fig. 1. We begin in Section 2 by introducing the `RunModel` module, which is at the core of `UQpy`, and enables almost all modeling activities – deterministic or probabilistic. Sections 3–6 are focused on the various capabilities in probabilistic modeling currently implemented in `UQpy`. This starts at the foundation of the software (bottom of Fig. 1) with the `SampleMethods` and `StochasticProcess` modules for simulation-based uncertainty propagation in Section 3. The more advanced methods (top of Fig. 1) are discussed in Sections 4–6. Section 4 deals with probabilistic inverse modeling through the `Inference` module. In Section 5, we present the `Reliability` module for rare-event simulation and probability of failure estimation. Finally, surrogate modeling and active learning capabilities are presented using the `Surrogates` module in Section 6. Two additional modules, the `Distributions` module for defining probability distribution objects and the `Transformations` module for transforming random variables are described in Appendices C and D. The `Utilities` module is not discussed in this paper. It is also important to mention that this manuscript is not intended to provide a detailed description of all `UQpy` functionalities and programming features. For this the reader is referred to the `UQpy` documentation [32]. It is aimed instead at providing an overview of the breadth of tasks that can be performed with this package, an introduction to its utilization and code framework, and a guide for researchers aiming to use it as a development tool. All the codes written within the manuscript are provided as supplementary materials, or can be downloaded from

GitHub [11].

Finally, in order to illustrate the various UQpy capabilities, we track two examples continuously throughout the paper. These two examples intentionally describe relatively simple models as they serve only to illustrate the various functionalities of UQpy. Much more complex models can be, and have been, integrated within this framework. One example is a structural dynamics model written completely in Python to demonstrate how UQpy works within the Python environment to execute Python models for UQ purposes. The second example is an Abaqus finite element model performing thermomechanical analysis of a beam. This example demonstrates the use of UQpy for models built using third-party software. These examples are introduced in Section 2 and are used throughout the paper.

2. Driving simulations: the RunModel module

2.1. Introduction

Many tasks in UQ, from forward uncertainty propagation to inverse learning, require running forward simulations of a computational model $h(\cdot)$ at various points in the space of input uncertainties X. The forward model can be generically represented as:

$$\mathrm{Y} = h(\mathrm{X}), \tag{1}$$

where Y is the output quantity of interest (QoI). In forward uncertainty propagation for instance, samples $\mathrm{x}^{(i)} \in \mathbb{R}^d, i = 1 : N$ are drawn from the known distribution $p(\mathrm{X})$ and the distribution of Y (or its moments) is inferred from outputs of the forward simulations $\mathrm{y}^{(i)} = h(\mathrm{x}^{(i)})$.

In UQpy, forward simulations are initiated via the RunModel module. This module can interact with Python computational models as well as third-party software, allowing great flexibility in the definition of the forward model $h(\cdot)$. If RunModel is used in combination with a Python computational model, the user must simply provide the filename of a '.py' file that contains the forward model function $h(\cdot)$. In this case, the model is directly imported into the user's Python environment and executed (it must be written in Python3). When running with a third-party software model, RunModel interfaces with the model through text-based input files and serves as the "driver" to initiate the necessary calculations. Examples of both types of applications are provided hereafter.

The jobs initiated by RunModel can be executed either in series or in parallel, allowing distribution of multiple jobs over multiple processes. The attribute ntasks is used to specify if series or parallel execution of jobs is desired. For parallelization across a single compute node or workstation, RunModel employs the multiprocessing Python package when run in combination with a Python computational model, and GNU parallel [34] when running a third-party software model. In the case of cluster computing with a third-party software model, RunModel uses GNU parallel to execute jobs in parallel over multiple cores on multiple compute nodes. This allows execution of jobs where the model is itself parallelized. But the supported options to request computational resources for such jobs are currently limited. Interested users are referred to the UQpy documentation [32] for more details. More support for execution of jobs on clusters will be rolled out in a future release. The software does not currently support parallel processing of Python models over multiple compute nodes.

Several examples for this paper, in particular those related to the structural fire example presented in Section 2.3, were executed using the Maryland Advanced Research Computing Center for third party (e.g. Abaqus) models requiring high performance computing.

An object of the class RunModel is instantiated as follows:

The minimum required and optional attributes of the RunModel

```
from UQpy.RunModel import RunModel
model = RunModel(model_script, input_template,
    ntasks, ...)
```

class (e.g., model_script, input_template, ntasks, ...) depend on the desired workflow – Python vs. third party-software and serial vs. parallel execution. Detailed examples of various workflows and the corresponding input attributes will be provided in the following sections. In order to instantiate one or several forward simulations, the run method is invoked as:

where samples is an array of at least two dimensions, with shape $(N, d, \ldots)$, that contains samples of the input random variables $\mathrm{x}^{(i)} \sim p(\mathrm{X})$. The convention adopted by RunModel is that the first index, N, is the number of samples and the second index, d, is the number of variables in each sample. Each of these d variables can be of arbitrary dimension, and hence samples can be of arbitrary dimension. N forward runs are then executed and the output quantities of interest $\mathrm{y}^{(i)}$ are stored as an attribute qoi_list (list of length N) of the RunModel object. The i^{th} sample $\mathrm{x}^{(i)}$ and corresponding QoI $\mathrm{y}^{(i)} = h(\mathrm{x}^{(i)})$ can thus be accessed as:

```
model.run(samples)
```

The run method can be invoked several times, in which case the

```
x_i = model.samples[i]
y_i = model.qoi_list[i]
```

samples and QoIs will by default be appended to existing values, unless input append_samples is set to False in which case existing values are overwritten. The user can also provide samples directly when instantiating the RunModel object, i.e.,

In this case, the run method is called during initialization and both

```
model = RunModel(samples, model_script,
    input_template, ntasks, ...)
```

the samples and QoIs are stored as previously described.

In the following sections, two examples are provided to illustrate the usage of the RunModel module and its various workflows. First, a simple Python model that solves the dynamics equation of a highly nonlinear single-degree-of-freedom system is presented. Then a more complex finite element model is described that illustrates the combination of RunModel with a third party software (Abaqus in this case).

2.2. Python computational model

This section illustrates how RunModel executes a Python computational model. The underlying physical problem deals with the dynamical behavior of a highly nonlinear single-degree-of-freedom (SDOF) system (see Fig. 3). The dynamical behavior of this Bouc-Wen model of hysteresis [35] can be represented by the following system of equations (the mass is assumed known as $m = 1$):

$$\ddot{z}(t) + c\dot{z}(t) + \mathrm{k}\ \mathrm{r}(t) = -\ddot{u}, \tag{2a}$$

$$\dot{r}(t) = \dot{z} - \beta|\dot{z}(t)||r(t)|^{n-1}r(t) - \gamma\dot{z}(t)|r(t)|^{n}, \tag{2b}$$

where z, r are the displacement and hysteresis variables respectively and $\ddot{u}$ is the ground acceleration that serves as input to the model. The model is parameterized by its stiffness k, damping c and Bouc-Wen parameters n, β, γ, which govern the shape of the hysteresis loop. Alternatively, the Bouc-Wen model can be reparameterized using the three parameters n, $r_0 = \sqrt[n]{\frac{1}{\beta+\gamma}}$, $\delta = \frac{\beta}{\beta+\gamma}$.

Uncertainties in this system originate from two distinct sources: stochasticity of the input ground acceleration, and randomness in the model parameters. Various UQ tasks can be performed with this model. Here, forward propagation of multiple sources of uncertainties, including high dimensional random vectors (stochastic processes), is illustrated in Section 3.2 and parameter learning and model selection from data are illustrated in Section 4. Two distinct models will be considered for these two tasks.

The first model considers input uncertainty in both the model parameters $[k, r_0, \delta, n]$ (assuming no damping, $c = 0$), and the input ground motion acceleration $\ddot{u}(t)$. The input X thus consists in five random variables ($d = 5$) of heterogeneous data types: four random scalars of model parameters and a stochastic process that models the input excitation. The quantity of interest Y is the time-dependent displacement response of the system. In `UQpy`, when defining a `RunModel` object that calls a Python computational model, the user must provide as input `model_script`, which is a string containing the name of the Python file that contains the model $h(\cdot)$. This Python model must be defined as either a class or a function with specific formatting rules, and the user is referred to the `UQpy` documentation [32] for more details. In the case considered herein, all Python functions related to this example are written in a single file 'utils_dynamics.py'. In particular, the function 'sdof_boucwen_prop' takes in as first input a sample consisting of one realization of the model parameters and one realization of the stochastic process,[1] solves the dynamics equations in Eq. (2) forward in time and returns the corresponding displacement. An abridged version of this function follows.

```
def sdof_boucwen_prop(samples, ...):
    # Compute displacement QoI for a sdof Bouc-
    Wen model, considering uncertainty in both
    model parameters and input excitation
    params = np.concatenate(samples[0, :4])
    input_acceleration = samples[0, -1]
    displacement = solve_dynamics(params,
    input_acceleration)
    return displacement
```

When defining the `RunModel` object, the name of the specific function within the `model_script` that executes the model should be provided as the `model_object_name` input. In this case, the `RunModel` object is then instantiated as follows:

```
dyn_model_prop = RunModel(model_script='
    utils_dynamics.py', model_object_name='
    sdof_boucwen_prop', var_names=['k', 'r0', '
    delta', 'n', 'input_accel'], ntasks=1, ...)
```

The input parameter `ntasks` is set to 1, specifying that execution of this model is to be performed serially (not in parallel). Setting ntasks to an integer greater than 1 would trigger the parallel workflow where ntasks are run concurrently.

The second model will be used for parameter estimation/model selection, and is thus defined to deal solely with uncertainties in the system parameters, i.e. $\mathrm{X} = [k, r_0, \delta, n]$ (assuming no damping, $c = 0$). The input excitation $\ddot{u}(t)$ is assumed known, it is a (scaled) version of the El-Centro earthquake ground motion, downloaded from [36]. The abridged function that computes the displacement QoI is written as follows:

```
def sdof_boucwen_infce(samples, scale_factor=1.,
    ...):
    # Compute displacement QoI for a sdof Bouc-
    Wen model, excitation is a scaled version of
    El-Centro ground motion.
    params = samples[0, :]
    input_acceleration = scale_factor *
    el_centro_data
    displacement = solve_dynamics(params,
    input_acceleration)
    return displacement
```

The above function takes in an additional parameter `scale_factor`, which can thus be fixed outside of the 'dynamics_utils.py' file. A value for this additional input must be provided when instantiating the `RunModel` object, as follows:

```
scale_factor = 0.1
dyn_model_infce = RunModel(model_script='
    utils_dynamics.py', model_object_name='
    sdof_boucwen_infce', var_names=['k', 'r0', '
    delta', 'n'], ntasks=4, scale_factor=
    scale_factor, ...)
```

The user can thus pass in any additional input that is used by the model $h(\cdot)$ that computes the QoI, allowing great flexibility in defining such models. Notice also that `ntasks=4` in this `RunModel` object, invoking parallel computing when the `run` method is called. Notice also that in this setting, the input uncertainty is assumed to be composed of four scalar random variables ($d = 4$), as indicated by the variable names `var_names`.

Both `RunModel` objects previously defined can then be used to execute the corresponding forward model, by calling the `run` method. The input `samples` must be such that `len(samples)`$=N$ and `len(samples[0])`$=d$. For instance, to run the second model for a given set of parameters $k = 1$ cN/cm, $r_0 = 2$ cm, $\delta = 0.9$ and $n = 3$, one calls:

```
samples = np.array([1.0, 2.0, 0.9, 3.0]).reshape
    ((1, 4))
dyn_model_infce.run(samples=samples)
```

The QoI for this one run is accessed via:

```
qoi = dyn_model_infce.qoi_list[0]
```

2.3. Third-party software computational model

This section presents how `RunModel` can be used to drive simulations of a model that is not in Python, with the help of an example. Such a model can be of any kind (e.g. commercial software or locally

[1] The `RunModel` module also supports vectorized computations for Python computational models; the model script would then accept several samples at once and return all associated QoIs.

compiled software) and requires only that the software provide an ASCII text-based input file through which the user defines the calculation and its parameters. In the example provided here, the deflection of a uniformly loaded beam subjected to fire-induced temperature change is simulated in the commercial finite element analysis software Abaqus [37]. The problem is detailed in Fig. 4, and the geometry is simplified from [38]. In particular, we investigate the influence of the fire load density and the room-temperature material yield strength (considering elastic-perfectly plastic material response) on the deflection of the midpoint of the beam as it undergoes a temperature change defined by the parametric curve [39] in Fig. 4, with a linear yield strength degradation curve also shown in Fig. 4. Given that the calculation is performed using numerical time integration, the relationship between the inputs (fire load density and yield strength) and outputs (deflection) cannot be expressed in closed form. Adopting the generic representation in Eq. (1), each execution of the model $h(\cdot)$, thus takes a sample of two uncertain scalar parameters as input X, and returns the difference between the maximum allowable displacement (d_a) and the maximum displacement of the midpoint of the beam (d_m) as the output, i.e., $Y = d_a - d_m$. The maximum allowable displacement is one of the criteria used to assess load-bearing capacity of the beam and is computed as $d_a = \frac{L^2}{400h}$, where L is the length of the beam, and h is the depth of the cross-section of the beam [40]. For the dimensions shown in Fig. 4, $d_a \approx 7.14$ cm.

The two required inputs when defining the `RunModel` object for this workflow are the name of the `model_script`, and the name of the `input_template`. In addition to these two required inputs, in this example, the name of an `output_script` is provided for post-processing. Unlike the Python workflow, the `model_script` is not the computational model itself. Instead, the `model_script` is a Python script (in the form of a function or class) with commands necessary to execute the third-party model. The `input_template` is a text file which contains placeholders demarcated by angle brackets $< \cdot >$ with the variable names inside. Standard Python indexing is supported inside the place-holders. `RunModel` scans the `input_template` file and replaces the placeholders with the corresponding sample values of the input variables. For example, the placeholder `<var[0][2]>` in the template file will cause the corresponding component of `var` to be placed at that location in the input file. For the computational model in the example, the variable names used are `qtd` for the fire load density, and `fy` for the yield strength. Since these are different from the default variable names used by `RunModel`, they have to be passed as an input when defining the `RunModel` object. Finally, the `output_script` will be executed by `RunModel` to retrieve the quantity of interest and save it in the attribute `qoi_list` of the `RunModel` object for postprocessing and adaptivity/learning.

The `RunModel` object is defined as follows:

```
abaqus_sfe_model = RunModel(model_script='
    abaqus_fire_analysis.py', input_template='
    abaqus_input.py', output_script='
    extract_abaqus_output.py', var_names=['qtd',
    'fy'], ...)
```

and can be used to execute the third-party software model by passing samples of the random variables as inputs to the `run` method as follows:

```
abaqus_sfe_model.run(samples=samples)
```

After execution of the model, the outputs corresponding to each sample are saved as a list in the attribute `qoi_list` of the `RunModel` object.

```
outputs = abaqus_sfe_model.qoi_list
```

This model will be used in later sections of the paper to demonstrate Monte Carlo simulation and the stratified sampling variance reduction method in Section 3.1.1. Reliability estimation capabilities of `UQpy` will be demonstrated with this model using FORM in Section 5.1, and using subset simulation in Section 5.2. Adaptive sampling capabilities in `UQpy` such as Adaptive-Kriging MCS will also be demonstrated using this model, in Section 6.2. Since this model is moderately computationally expensive, a surrogate model will be trained to represent the performance function from this model, in Section 6.3.

3. Forward propagation of uncertainties

3.1. *Sampling random variables: the* `SampleMethods` *module*

In forward simulation, one wants to study how uncertainties in inputs X, with known probability density $p(\mathrm{X})$, propagate through the computational model $h(\cdot)$ and affect the output quantities of interest Y. Propagation of uncertainties generally requires evaluating the computational model at various points $\mathrm{x}^{(i)}$ of the input space. The `SampleMethods` module contains several classes to sample realizations $\mathrm{x}^{(i)}$ of random variable (RV) X. Methods such as simple Monte Carlo sampling (MCS) are available, along with variance reduction sampling techniques such as Latin Hypercube Sampling (LHS) [41,42], stratified sampling (STS) [41], and some adaptive variations of these (e.g. refined stratified sampling, RSS) [43,44], which require the distribution of the RVs to be known in advance. The module also contains classes to sample from distributions that are known up to a constant, such as Markov Chain Monte Carlo algorithms and Importance Sampling.

Sampling schemes available in the `SampleMethods` module, such as `MCS`, `LHS`, `STS`, and `RSS`, can be used to generate independent random draws from a specified probability distribution or distributions. The `Distributions` module of `UQpy` (Appendix C) is utilized by the `SampleMethods` module to define probability distributions. This module includes a variety of univariate and multi-variate distributions; it also supports user-defined distributions and multi-variate distributions defined via their marginals, possibly with a copula to introduce dependence between components. Another option to induce correlation between the components of random vectors is to use the `Transformations` module, described in Appendix D.

In the following sections, we describe the primary sampling methods that are available in `UQpy`. The emphasis here is placed on their implementation in the software and their use for uncertainty propagation. For this reason, we do not specifically describe each method in detail. For such descriptions, references are provided. Additionally, we further discuss several of these methods in the following section in the context of probabilistic inverse problems. Again, here the focus is on forward propagation.

3.1.1. *Combining* `SampleMethods` *and* `RunModel` *to propagate uncertainties*

For general uncertainty propagation, samples generated using the `SampleMethods` module can be passed as inputs to the `run` method of a `RunModel` object. To illustrate this point, we consider two different sampling schemes to generate random inputs for the third-party Abaqus thermomechanical model described in Section 2.3. Consider that the fire load density is modeled as a uniformly distributed random variable on the range 50–450 MJ/m^2 and the yield strength at room temperature is modeled as a normally distributed random variable with a mean value of 250 MPa and a coefficient of variation of 7%. To draw samples from these distributions using any of the sampling schemes available in UQpy, distribution objects are first created:

```
from UQpy.Distributions import Normal, Uniform
dist1 = Uniform(loc=50, scale=400)
dist2 = Normal(loc=250e6, scale=17.5e6)
dists = [dist1, dist2]
```

Samples of these parameters can be drawn using Monte Carlo sampling (the `MCS` class) as follows:

```
from UQpy.SampleMethods import MCS
x_mcs = MCS(dist_object=dists, random_state
    =1234567890)
x_mcs.run(nsamples=1024)
```

Similarly, samples of these random variables can be drawn using stratified sampling (the `STS` class) as follows:

```
from UQpy.SampleMethods import RectangularStrata,
     RectangularSTS
strata = RectangularStrata(nstrata=[32, 32])
x_sts = RectangularSTS(dist_object=dists,
    strata_object=strata, random_state
    =1234567890)
x_sts.run(nsamples_per_stratum=1)
```

In each case, we draw 1024 samples. For `MCS` these samples are randomly drawn according to the given distributions. For `STS`, the two-dimensional domain is discretized into 32×32 disjoint square strata of equal probability and one sample is drawn from each stratum. `UQpy` supports several types of geometric stratifications (rectangular, voronoi, delaunay), and user-defined stratifications can easily be implemented via sub-classing. The `random_state` input allows the user to seed the pseudo-random number generator and obtain reproducible results. This `random_state` input is included wherever needed in the code and used throughout the examples presented in the manuscript.

The third-party model can be executed for each of these sample sets as follows, for `MCS`:

```
abaqus_sfe_model.run(samples=x_mcs.samples)
```

For `STS`:

```
abaqus_sfe_model.run(samples=x_sts.samples)
```

Recall that the `RunModel` object `abaqus_sfe_model` has been previously initialized in Section 2.3.

Fig. 5 shows the samples generated using both `MCS` and `STS`. The finite element model was evaluated at these points to calculate the maximum deflection of the midpoint of the beam. This deflection is compared to a deflection tolerance of 7.14 cm, and the sample markers in Fig. 5 are colored blue if the computed deflection does not exceed this threshold, and are colored red if the deflection exceeds the threshold. In the latter case, the beam is considered to have failed. Therefore, propagating samples through the model in this manner allows estimation of the probability of failure of the beam, which is estimated by Monte Carlo simulation to be 3.3% using 10,000 samples (not shown). We will return to this problem of failure probability estimation in the context of reliability analysis in Section 5.

3.1.2. Markov chain Monte Carlo algorithms and importance sampling

The `SampleMethods` module of `UQpy` also includes algorithms for Markov chain Monte Carlo (MCMC) and importance sampling (IS). These algorithms are used to sample from probability distributions $p(\mathrm{X})$ that are either difficult to sample from or may only be known up to a constant, i.e., $p(\mathrm{X}) = \frac{\tilde{p}(\mathrm{X})}{C}$, where $\tilde{p}(\mathrm{X})$ can be evaluated but C is unknown – as is the case when using Bayes' theorem for instance. The `MCMC` and `IS` classes can be used as stand-alone tools for sampling from $p(\mathrm{X})$, but they are also invoked by other classes in `UQpy` to perform advanced UQ tasks, such as Bayesian estimation (`Inference` module) or estimation of failure probability via Subset Simulation (`Reliability` module). These are specifically discussed in Sections 4 and 5, respectively.

MCMC algorithms build a Markov chain that has the desired target distribution $p(\mathrm{X})$ as its equilibrium distribution, thus states of the chain are samples of the desired distribution. There are numerous MCMC algorithms, and a comprehensive review of these algorithms is beyond our scope. The reader is referred to e.g. [45] for an introduction to some MCMC algorithms and [46] for more theory about MCMC methods. Currently, `UQpy` includes the following MCMC algorithms, where each algorithm is implemented as a class that inherits from a parent `MCMC` class:

- Metropolis-Hastings (MH): (`MH` class) The most well-known algorithm, MH samples a candidate $\mathrm{x}^\star$ from a previous state x_{k-1} using a user-defined proposal distribution $\mathrm{x}^\star \sim J(\cdot|\mathrm{x}_{k-1})$ and accepts it with probability $\alpha = min\left\{\frac{\tilde{p}(\mathrm{x}^\star)}{\tilde{p}(\mathrm{x}_{k-1})}\frac{J(\mathrm{x}_{k-1}|\mathrm{x}^\star)}{J(\mathrm{x}^\star|\mathrm{x}_{k-1})}, 1\right\}$. In `UQpy`, the user can run several chains in parallel by providing multiple seed points.
- Component-wise Modified Metropolis-Hastings (MMH): (`MMH` class) In MMH [47], sample components are accepted or rejected according to the MH acceptance/rejection scheme in one dimension at a time. If the target pdf can be factorized into a product of one-dimensional distributions, the MMH can be used.
- Delayed Rejection Adaptive Metropolis (DRAM): (`DRAM` class) The DRAM method [48] combines the delayed-rejection principle with adaptation of the proposal covariance matrix in an MH acceptance/rejection scheme.
- DiffeRential Evolution Adaptive Metropolis (DREAM) (`DREAM` class) The DREAM algorithm [49,50] runs several MH chains simultaneously and automatically tunes the scale and orientation of the proposal distribution in randomized subspaces during the search.
- Affine Invariance Ensemble Sampler with Stretch Moves (Stretch) (`Stretch` class) The Stretch sampler [51,52] leverages an affine-invariant property in a scheme that propagates an ensemble of walkers.

In the interest of brevity, all of the available algorithms will not be illustrated here.

In `UQpy`, the target distribution $p(\mathrm{X})$ is defined as a callable that computes $\tilde{p}(\mathrm{x})$ or $ln\tilde{p}(\mathrm{x})$ (the latter is preferred for stability reasons) for a given ndarray x of shape (N,d). Whenever possible, the log pdf of several samples is evaluated at once, for example when several chains are run in parallel. The target function callable can also take in any number of positional arguments. For instance, in order to sample from a 2D Rosenbrock function, the following callable is created:

```
def log_pdf_target(x, param):
    return -(100*(x[:, 1]-x[:, 0]**2)**2+(1-x[:,
    0])**2)/param
```

The target function callable is provided to the `MCMC` class as input `pdf_target` or `log_pdf_target`. All additional positional arguments are provided as a tuple in input `args_target`, the special Python syntax `*args` is used within the code to transfer these positional

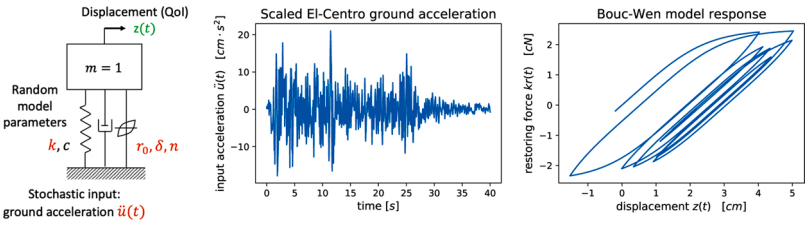


Fig. 3. Dynamics example: Left: physical system and sources of uncertainty; middle: El Centro ground acceleration as input excitation; right: system response for a given realization of the input excitation and system parameters.

arguments to the target callable. Any `MCMC` class takes in several additional arguments such as `nburn` (the number of samples to discard for burn-in), `jump` (the thinning parameter), and `seed` (the starting state (s)). Certain inputs are algorithm-specific; the MH algorithm for instance utilizes a user-defined proposal distribution. The user is referred to the `UQpy` documentation [32] for a more complete description of the input arguments to the various `MCMC` classes.

To sample from the above Rosenbrock distribution using the Metropolis-Hastings algorithm, the MCMC sampler is initialized and run as follows:

```
# Additional positional arguments of the target
    callable
args_rosenbrock = (20, )
# Define the proposal distribution
proposal = JointInd(marginals=[Normal(scale=0.5),
     Normal(scale=2.)])
# Create the MH object and sample
from UQpy.SampleMethods import MH
sampler = MH(log_pdf_target=log_pdf_target,
    args_target=args_rosenbrock, dimension=2,
    nchains=1, nburn=500, jump=100, proposal=
    proposal, ...)
sampler.run(nsamples=500)
```

Input `nsamples` represents the number of samples saved by the algorithm after discarding burn-in and thinning. If the user provides `nsamples` when initializing the class, the `run` method is directly called at initialization. Fig. 6a shows the resulting samples drawn from this Rosenbrock function using the MH algorithm.

The `MCMC` class was developed in such a way that new advanced algorithms can be quite easily integrated within the existing framework, thus allowing researchers to implement their novel algorithms, compare them with existing methods, and make them available to the research community. As previously mentioned, each MCMC algorithm is implemented as a class that inherits from a parent `MCMC` class. This parent class initializes the generic inputs that are shared across all algorithms, such as the target pdf, burn-in and so on. It defines the main `run` method that runs the chain forward and stores the samples; this method is also shared across all algorithms. The `run` method relies on a `run_one_iteration` method to run one state of the chain: it takes in as inputs the current state `current_state` and its log-pdf `current_log_pdf` and returns the new state and its log-pdf value. This `run_one_iteration` method contains the core MCMC algorithm, and it is thus being over-written by each new subclass that codes a specific algorithm. Adding a new algorithm can therefore be easily done – as illustrated schematically in the code section below – by defining a new class that inherits from `MCMC`, initializing any algorithm-specific input within the _init_ function and over-writing the `run_one_iteration` method that propagates the chain forward. This setup avoids any interaction with the existing code.

```
# Existing base MCMC class, defines generic
    attributes and methods
class MCMC:
    def __init__(self, pdf_target, nburn, ...):
        # initialize generic inputs
    def run(self, nsamples):
        # run iterations and save samples
        for i in range(nsamples):
            current_state =
                self.run_one_iteration(
                current_state)
            self.samples[i] = current_state

# User-defined MCMC sampler
class NewMetropolis(MCMC):
    def __init__(self, pdf_target, nburn,
    new_input, ...):
        # initialize generic and algorithm
        # specific inputs
        super().__init__(pdf_target, nburn, ...)
        custom_initialization(new_input)
    def run_one_iteration(self, current_state):
        # run one iteration of new Metropolis
        # algorithm, compute new state
        return new_state
```

Finally, one can also sample from a distribution that is known only to a scale factor ($p(\mathrm{X}) = \frac{\tilde{p}(\mathrm{X})}{C}$) via self-normalized importance sampling (IS). In IS, one draws samples $\mathrm{x}^{(i)}$ from a proposal distribution $\pi(\mathrm{X})$ that is easy to sample from, then weights the samples to account for the discrepancy between the sampling and target distributions. The weights are computed as $w^{(i)} = \tilde{p}(\mathrm{x}^{(i)})/\pi(\mathrm{x}^{(i)})$, then normalized to sum up to 1 so that the weighted set of samples defines an appropriate probability distribution. In `UQpy`, the user defines the target pdf with inputs `log_pdf_target` or `pdf_target` as in the `MCMC` class. The proposal must be provided as a `Distribution` object that has an `rvs` method and a `log_pdf` or `pdf` methods. Fig. 6b shows an importance sampling estimation of the Rosenbrock function, obtained via the following code:

```
proposal = JointInd(marginals=[Uniform(loc=-4.5,
    scale=11.), Gamma(a=1.5, loc=-1, scale=8.)])
from UQpy.SampleMethods import IS
sampler = IS(log_pdf_target=log_pdf_target,
    args_target=args_target, proposal=proposal)
sampler.run(nsamples=4000)
```

Note that IS is also used as a popular variance reduction scheme that is commonly employed in reliability analysis. The `IS` class can be easily integrated in IS-based reliability methods, although this is not explicitly illustrated here.

3.2. Simulation of stochastic processes and random fields: the `StochasticProcess` module

3.2.1. The `StochasticProcess` module

A stochastic process, or random field, is defined as a set of random variables defined on an indexed set. When the indexed set is defined as points in time, the stochastic processes describes the stochastic evolution of a time-dependent system. When the indexed set represents points on the Euclidean space, stochastic processes are more commonly referred to as random fields. Here, we use the term stochastic process to represent both since they are mathematically equivalent. Stochastic processes find applications in numerous disciplines ranging from biology, physics, neuroscience, signal processing, finance, and statistical mechanics, to computational mechanics which is the context here. Stochastic processes are especially useful in the analysis of complex nonlinear systems where simulation is essential for analysing the system. Simulation of stochastic processes generally utilizes an expansion of the following form:

$$A(t) = \sum_{i=1}^{M} C_i(\omega)\theta_i(t), \tag{3}$$

where $C_i(\omega)$ are a set of random variables and $\theta(t)$ are deterministic basis functions. Arguably, the two most widely used methods for the simulation of stochastic processes are the spectral representation method (SRM) [53,54] and the Karhunen-Loeve expansion (KLE) [55,56], both of which are implemented in `UQpy`. The two methods differ in their prescribed basis $\theta(t)$. The SRM utilizes a Fourier basis while the KLE derives its basis as the eigenfunctions of the covariance function.

The existing implementations of the SRM and KLE, given in the `SRM` and `KLE` classes, are used to generate Gaussian stochastic processes in which the random variables in the SRM are independent random phase angles uniformly distributed on $[0, 2\pi]$, and in the KLE the random variables are independent standard normal.

For the simulation of non-Gaussian processes, `UQpy` provides two methods. The first method simulates translation processes [57] in which an underlying Gaussian power spectral density (for SRM) or covariance function (for KLE) and a marginal non-Gaussian distribution are defined. In `UQpy`, this is performed using the `Translation` class. When defining a translation process, it is common to define the non-Gaussian power spectral density (or covariance function) and the marginal non-Gaussian distribution. However, simulation of translation processes requires an underlying Gaussian power spectral density/covariance function. To identify this underlying Gaussian process, `UQpy` utilizes the iterative translation approximation method (ITAM) [58] to overcome the issue of translation process incompatibility. The `InverseTranslation` class performs this iterative algorithm and can be used in support of simulation of non-Gaussian translation processes utilizing either the SRM or KLE.

The second method, referred to as the bispectral representation method (BSRM) and implemented in the `BSRM` class, is the 3rd-order generalization of the SRM derived in [59] in which the stochastic process is expanded from both the power spectrum and the bispectrum of the process.

`UQpy` also supports the simulation of Gaussian multi-dimensional random fields and multi-variate random processes using the `SRM` class as well as multi-dimensional non-Gaussian random fields using the `BSRM` class. Non-Gaussian multi-dimensional random fields and multi-variate random processes can be simulated using the `Translation` class. In future releases we anticipate extending the capabilities for simulation of Gaussian and non-Gaussian and non-stationary random processes, as well as introducing a new class for the simulation of Gaussian and non-Gaussian stochastic waves [60].

Here, we illustrate how to simulate a 1-dimensional, uni-variate stationary Gaussian random process using the `SRM` class and then show how to simulate a 1-dimensional, uni-variate stationary non-Gaussian random process using the `InverseTranslation` and `KLE` classes.

To simulate using the `SRM`, the user must specify the discretized power spectral density as well as the time and frequency discretizations. For the examples shown here, the power spectrum and discretizations are given by:

$$\begin{aligned} S\left(\omega\right) &= \frac{130}{4}\omega^2 e^{-5\omega}, \\ \Delta\omega &= 0.01, \\ \omega_u &= 1.28\text{rad/sec}, \end{aligned} \tag{4}$$

where S is the power spectrum, $\Delta\omega$ is the frequency discretization and ω_u is the upper cutoff frequency. An abridged code to simulate 1000 realizations of this random process follows:

```
from UQpy.StochasticProcess import SRM
SRM_object = SRM(nsamples=1000, power_spectrum=S,
     time_interval=dt, frequency_interval=dw,
    number_time_intervals=nt,
    number_frequency_intervals=nw, random_state
    =1234)
samples = SRM_object.samples
```

where `S` is an array containing the discretized power spectrum. The specified power spectral density and two sample realizations of the random process are shown in Fig. 7a and b.

For simulation with the `KLE` class, the user must specify the discretized autocorrelation function and the length of the time discretization. Additionally, the number of eigenvalues to be used in the expansion can be specified as well. For brevity, this case will not be illustrated here.

Next, we translate the samples along with the power spectrum of the Gaussian process to a lognormal process using the `Translation` class. The translated lognormal power spectrum is shown along with the Gaussian power spectrum in Fig. 7a and samples are plotted in Fig. 7c. Subsequently, we re-identify the underlying Gaussian power spectrum which, upon translation, would yield the lognormal power spectrum using the `InverseTranslation` class (i.e. using the ITAM). The actual Gaussian and the identified Gaussian power spectra are plotted in Fig. 8. As we can see, these power spectra align nearly perfectly.

A brief outline of the code to execute the translation and inverse translation is shown below.

```
from UQpy.StochasticProcess import Translation,
    InverseTranslation
from UQpy.Distributions import Lognormal

dist_object = Lognormal(0.5, 0, np.exp(0.5))
Translate_object = Translation(dist_object=
    dist_object, time_interval=dt,
    frequency_interval=dw, number_time_intervals=
    nt, number_frequency_intervals=nw,
    power_spectrum_gaussian=S, samples_gaussian=
    samples)
samples_ng = Translate_object.
    samples_non_gaussian
S_ng = Translate_object.
    power_spectrum_non_gaussian
R_ng = Translate_object.
    correlation_function_non_gaussian

Inverse_translate_object = InverseTranslation(
    dist_object=dist_object, time_interval=dt,
    frequency_interval=dw, number_time_intervals=
    nt, number_frequency_intervals=nw,
    correlation_function_non_gaussian=R_ng,
    samples_non_gaussian=samples_ng)
R_g_inv = Inverse_translate_object.
    correlation_function_gaussian
S_g_inv = Inverse_translate_object.
    power_spectrum_gaussian
```

3.2.2. *Propagation of heterogeneous uncertainties using* `RunModel`

This section illustrates how to propagate multiple sources of uncertainties, including random variables and high-dimensional stochastic processes, through a computational model with `RunModel`. Recall that the `RunModel` object for propagation of combined model parameters and input excitation uncertainties through a nonlinear SDOF dynamical system has been created in Section 2. To propagate uncertainties, the user must simply call the `run` method, giving to it as input the samples composed of both realizations of random model parameters and realizations of the stochastic process generated previously via SRM. Here, we emphasize that `RunModel` requires that samples have a shape of (N, d) where N is the number of samples and d is the number of variables. However, each variable need not be a scalar. In the example provided here, the samples are passed in as five variables (`k, r0, delta, n, accel`) where the first four variables are scalars and the final variable is an array containing the generated random process. This convention allows `UQpy` to employ standard Python indexing for high-dimensional variables. For example, within the calls to the computational model executed by `RunModel` one could extract a specific component of the variable `accel`. Abridged code is provided below for simulations with both Gaussian and lognormal excitation. Sample dynamic responses are presented in Fig. 9.

```
# Generate realizations of model parameters, for
    instance using MCS
samples_k, samples_r0, samples_delta, samples_n =
     MCS(...)
# Run model for Gaussian excitation, overwrite
    all previous samples and qois
samples = [[k, r0, delta, n, accel] for k, r0,
    delta, n, accel in zip(samples_k, samples_r0,
     samples_delta, samples_n, samples_g)]
dyn_model_prop.run(samples=samples,
    append_samples=False)
# Run model for log normal excitation, overwrite
    all previous samples and qois
samples = [[k, r0, delta, n, accel] for k, r0,
    delta, n, accel in zip(samples_k, samples_r0,
     samples_delta, samples_n, samples_ng)]
dyn_model_prop.run(samples=samples,
    append_samples=False)
```

4. Probabilistic inverse learning: the `Inference` module

4.1. *Introduction and structure*

The goal in inference can be twofold: given some data D, estimate the parameters of a model and/or assess the performance of a set of candidate models (i.e. model selection). `UQpy` supports various algorithms for parameter estimation and model selection, as summarized in Fig. 10.

In the following, it is assumed that the probabilistic model for inference is of the form:

$$D \sim h(\mathbf{X}) + \epsilon, \tag{5}$$

where $h(\mathrm{X})$ is a parametric computational model with parameters X that is executed via `RunModel` and the error ε is assumed to be Gaussian with zero mean. D in this case is a one-dimensional `numpy` ndarray of shape $(n_{\mathrm{D}},)$. `UQpy` supports a wider variety of inference problems, such as non-Gaussian error models, learning the parameters of a probability distribution defined by an object of the `Distribution` class, or problems specified by a user-defined likelihood function. Due to limited space however, the present manuscript focuses on the generic problem defined by Eq. (5), and the interested reader is referred to the `UQpy` documentation [32] for details about the more advanced capabilities of the `Inference` module. It is to be noted though that, at the time of this article, `UQpy` only supports off-line methods for inference, i.e., the whole data set must be provided up-front. Sequential methods may be considered for future release.

A model in the form of Eq. (5) is defined in the `Inference` module by an object of the `InferenceModel` class. In the `InferenceModel` class, the computational model $h(\cdot)$ is defined as a parameterized `RunModel` object, ε is specified by its covariance, and (for Bayesian methods) a prior for the parameter vector is specified as an object of the `Distribution` class. The main role of the `InferenceModel` object is to evaluate the log likelihood function of the data for the model, $lnp(\mathrm{D}|\mathrm{x}^{(i)}), i = 1:N$ or the scaled log posterior, $lnp(\mathrm{D}|\mathrm{x}^{(i)})p(\mathrm{x}^{(i)}), i = 1:N$ for Bayesian estimation. To do so, the `InferenceModel` possesses an `evaluate_log_likelihood` method (`evaluate_log_posterior` for Bayesian estimation), which takes as inputs a data vector D denoted `data` and a 2D `numpy` ndarray `params` of N parameter vectors $\mathrm{x}^{(i)}, i = 1:N$. This calculation involves simulation of the model via `RunModel` and leverages its parallel execution when $N > 1$. The `evaluate_log_likelihood` method (`evaluate_log_posterior` for Bayesian estimation) is at the core of the `Inference` module; it is invoked by all its remaining classes to perform parameter learning and model selection. Although discussed here in the context of a Gaussian error, we further emphasize that, for more general cases, the user can provide a custom non-Gaussian log-likelihood function.

In the following sections, the capabilities of the `Inference` module will be illustrated on the dynamics computational model presented in Section 2 (`dyn_model_infce` `RunModel` object). The data D used for inference consists of a displacement time-series of total duration 40 s, sampled at 50 Hz. This data was generated synthetically from a Bouc-Wen model with $k = 1$ cN/cm, $r_0 = 2.5$ cm, $\delta = 0.9$, $n = 3$, and small viscous damping, thus introducing some modeling error since the model used for inference assumes no damping. 5% root-mean-square (RMS) Gaussian noise was also added to the data to simulate measurement noise. The vector of parameters to be estimated is thus $\mathrm{X} = [k, r_0, \delta, n]$, with prior defined as follows (shown in Fig. 11):

```
var_names = ['k', 'r0', 'delta', 'n']
from UQpy.Distributions import JointInd, Uniform,
     Lognormal
prior = JointInd(marginals=[Uniform(loc=0.5,
    scale=2.5), Lognormal(s=0.5, loc=1., scale
    =2.), Lognormal(s=0.8, loc=0.5, scale=2.),
    Uniform(loc=1.1, scale=8.9)])
```

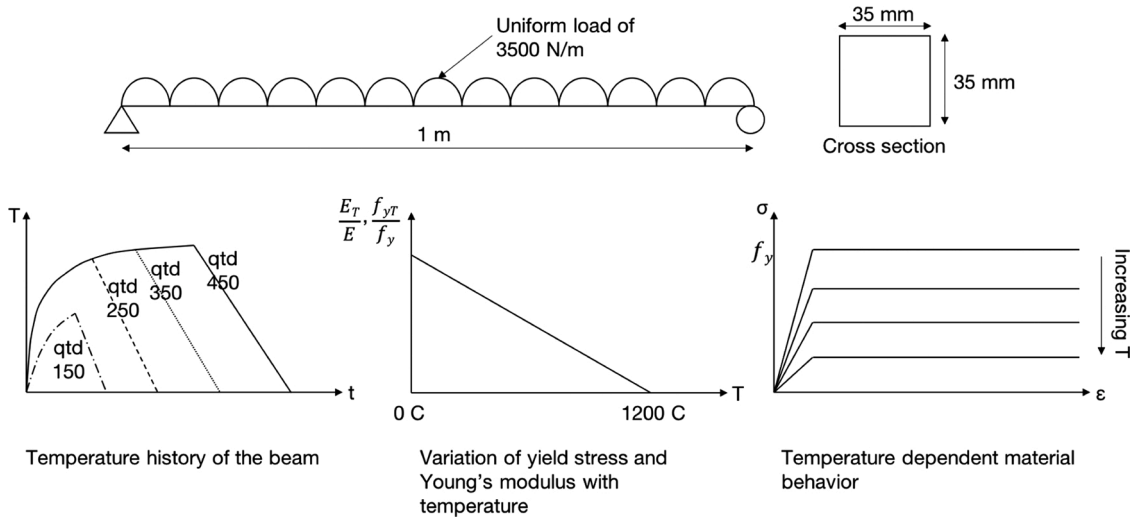


Fig. 4. Structural fire engineering example: Top – simply supported beam with uniform load and elevated temperature. Bottom from left to right – sample temperature histories defined by parametric fire curves, temperature-dependence of the yield strength and Young's modulus, and temperature-dependent elastic perfectly plastic stress-strain relations.

The `InferenceModel` object is then created as:

```
from UQpy.Inference import InferenceModel
inf_model_bw = InferenceModel(nparams=4,
    runmodel_object=dyn_model_infce,
    error_covariance=variance_noise, prior=prior,
     name='BoucWen')
```

where `variance_noise` is assumed to be known as 5% of the RMS of the data. Using this `InferenceModel` object, we now turn our attention to parameter estimation for the model.

4.2. Parameter estimation

Parameter estimation techniques aim to determine the parameters governing a computational model based on observed noisy data, and quantify uncertainties associated with these parameters due to measurement errors. The interested reader is referred to e.g. [45] for a thorough introduction to parameter estimation using frequentist and Bayesian approaches. In this section, we describe the two forms of parameter estimation available in `UQpy`: maximum likelihood estimation and Bayesian parameter estimation.

4.2.1. Maximum likelihood estimation: the `MLEstimation` class

In a frequentist approach, the parameter value that makes the measured data most likely is the maximum likelihood estimate:

$$\mathrm{x}_{\mathrm{ML}} = \mathrm{argmax}_{\mathrm{X}} \quad p(D|\mathrm{x}). \tag{6}$$

In `UQpy`, the `MLEstimation` class of the `Inference` module computes this maximum likelihood (ML) estimate. The `MLEstimation` class operates on an `InferenceModel` object (described above) and a data vector to maximize the likelihood function using a specified optimizer (default is `scipy.optimize.minimize`). The user can also leverage advanced global optimization functions by providing an optimizer function as input `optimizer` to the `MLEstimation` class. The reader is referred to the `UQpy` documentation [32] for details about the requirements of this optimizer function; the example that follows leverages the basin-hopping global optimization scheme (see `scipy.optimize.basinhopping`).

To obtain the maximum likelihood parameter estimates for the dynamics problem previously presented, an `MLEstimation` object is created as follows:

```
from UQpy.Inference import MLEstimation
ml_estimator = MLEstimation(inference_model=
    inf_model_bw, data=data_noisy, optimizer=
    basinhopping, niter_success=10, ...)
```

where `niter_success` is an input parameter of the `scipy.optimize.basinhopping` function. The optimization procedure is instantiated by providing an initial guess `x0` to the `run` method[2] as follows:

```
ml_estimator.run(x0=[1.7, 3.2, 3.3, 5.5])
```

The values of the fitted parameters and the value of the maximum log likelihood are stored as attributes of the object, `ml_estimator.mle` and `ml_estimator.max_log_like` respectively. For this example, the MLE is given by:

```
print(ml_estimator.mle)
>> [0.9987461  2.65146952 1.03224652 2.65229821]
```

4.2.2. Bayesian parameter estimation: the `BayesParameterEstimation` class

In the Bayesian paradigm, parameters are treated as random variables with associated pdfs that represent the state of knowledge about that parameter. The prior pdf $p(\mathrm{X})$ incorporates information available to

[2] For all inference classes, if `x0` (or `nsamples` in Bayesian estimation) is provided when creating the object, the `ml_estimation` is directly called when instantiating the object.

the user prior to observing data. It is updated upon observing data D using Bayes' theorem to yield the posterior pdf as:

$$p(\mathrm{X}|D) = \frac{p(D|\mathrm{X})p(\mathrm{X})}{p(D)}. \tag{7}$$

In `UQpy`, the `BayesParameterEstimation` class draws samples from the posterior pdf using MCMC or IS by calling an `MCMC` or `IS` class from the `SampleMethods` module. The `BayesParameterEstimation` class leverages the `InferenceModel` object and a data vector to compute the scaled posterior (numerator of Eq. (7)), and uses this in the specified `sampling_class` (`IS` or an `MCMC` subclass, i.e. `MH`, `Stretch`, ...) to draw samples from the posterior. For the model considered herein, the `BayesParameterEstimation` object is created as follows:

```
from UQpy.Inference import
    BayesParameterEstimation
from UQpy.SampleMethods import Stretch
be = BayesParameterEstimation(data=data_noisy,
    inference_model=inf_model_bw, sampling_class=
    Stretch, seed=seed, scale=2)
```

where `seed` and `scale` are inputs to the `Stretch` class. In this example, the affine-invariant ensemble sampler with stretch moves is utilized for MCMC, starting with 16 walkers initially drawn in a region near the ML estimate.

Samples from the posterior are drawn by calling the `run` method as follows:

```
be.run(nsamples=5000)
```

which simply calls the `run` method of the `MCMC` (or `IS`) sampler. The user can thus continue drawing new samples by calling the `run` method several times.

Outputs of the class `BayesParameterEstimation` are samples from the posterior pdf (weighted samples in the case of IS). Results of the Bayesian estimation for the dynamics model are shown in Fig. 11, highlighting the uncertainties in the model parameters inferred from the data.

4.3. Model selection

4.3.1. Problem statement

Model selection refers to the task of statistically selecting a model from a set of candidate models, given some data. In the following, the noisy data previously described is used to identify the 'correct' dynamics equation governing the SDOF system from noisy displacement data. The candidate models are:

- a linear model governed by a stiffness parameter k [cN/cm] and viscous damping parameter c [cN/m][3] ;
- an elastic-perfectly plastic model, parameterized by $\mathrm{X} = \{k, z_p, c\}$ where k is the stiffness parameter, z_p [cm] is the yield displacement, and c is the damping coefficient;
- a Bouc-Wen model of hysteresis without damping as previously described, parameterized by $\mathrm{X} = \{k, r_0, \delta, n\}$.

Model selection can be performed by minimizing a chosen information theoretic criterion, or in a Bayesian fashion by computing the posterior probability of each candidate model. The interested reader is referred to e.g. [61] for more theory and applications in stochastic dynamics. `UQpy` supports both approaches for model selection, however in its current version the Bayesian model selection class uses a simplistic formula for computation of the model evidence, which needs to be improved upon to yield more reliable results. For the problem presented herein, only the information theoretic approach will be illustrated.

Model selection methods in `UQpy` require the user to provide a list of candidate models, where all candidate models are objects of the `InferenceModel` class. For the problem at hand, the linear and elastoplastic models must be defined in addition to the Bouc-Wen model previously studied as follows:

```
# Define the elastic inference model
dyn_linear = RunModel(model_script='
    utils_dynamics.py', model_object_name='
    sdof_linear_infce', var_names=['k', 'c'],
    scale_factor=scale, ...)
inf_model_linear = InferenceModel(nparams=2,
    runmodel_object=dyn_linear, error_covariance=
    variance_noise, name='linear')
# Define the elastoplastic inference model in a
    similar fashion
dyn_elastoplastic = RunModel(...)
inf_model_elastoplastic = InferenceModel(
    runmodel_object=dyn_elastoplastic, ...)
# Define the list of candidate models
candidate_models = [inf_model_linear,
    inf_model_elastoplastic, inf_model_bw]
```

4.3.2. Information theoretic model selection: the `InfoModelSelection` class

In this approach, an information theoretic criterion is computed for all candidate models. The model that minimizes the chosen criterion is selected as the 'best' model given the data. `UQpy` implements three criteria – the Bayesian information criterion (BIC) [62], the Akaike information criterion (AIC) [63], and the Akaike criterion with correction for small data sets (AICc) [64,65]. The AIC for instance is defined as:

$$\mathrm{AIC} = \underbrace{-2ln\widehat{L}}_{\substack{\text{data}\\\text{fit term}}} + \underbrace{2d}_{\substack{\text{penalty against}\\\text{model complexity}}}, \tag{8}$$

where d is the number of parameters characterizing the model and $\widehat{L} = p(\mathrm{D}|\mathrm{x}_{\mathrm{ML}})$ is the maximum value of the likelihood function.

In `UQpy`, this procedure is performed using the `InfoModelSelection` class. It takes as required inputs a list of `InferenceModel` objects and a data vector. The `InfoModelSelection` class leverages the `MLEstimation` class to perform maximum likelihood estimation for each model and compute $\widehat{L}$. Inputs to `MLEstimation` are provided to `InfoModelSelection` as lists of length equal to the number of models. For the example at hand, the `InfoModelSelection` object is created as follows:

```
from UQpy.Inference import InfoModelSelection
optimizer = [basinhopping] * 3
niter_success = [15, 15, 10]
selector = InfoModelSelection(candidate_models=
    candidate_models, data=data_noisy, criterion=
    'AIC', optimizer=optimizer, niter_success=
    niter_success, ...)
```

Creating this object also instantiates a `MLEstimation` object for each model, stored in a list `selector.ml_estimators`.

The model selection procedure is performed when calling the `run` method of the `InfoModelSelection` object as follows:

[3] The unit cN/m is chosen for the damping parameter so as to keep all parameters in the same order of magnitude.

```
x0 = [[1.7, 1.], [1.7, 3.2, 1.], [1.7, 3.2, 3.3,
    5.5]]
selector.run(x0=x0)
```

The lists of output criterion values and model probabilities are stored as attributes `selector.criterion_values` and `selector.probabilities` respectively. In the present case, the results of the model selection procedure are as follows:

```
>> Model linear: AIC value = 821888, probability
    = 0.00
>> Model elastoplastic: AIC value = 1783,
    probability = 0.00
>> Model BoucWen: AIC value = -2402, probability
    = 1.00
```

The model that minimizes the AIC criterion is the Bouc-Wen model of hysteresis, which was expected as it was used to synthetically generate the data. This can also be qualitatively assessed by running simulations of the three systems with the fitted ML parameters, stored as attributes of the `MLEstimation` objects. For instance, the ML estimate of the first model (linear) can be accessed as:

```
mle_linear = selector.ml_estimators[0].mle
```

Fig. 12 illustrates this model comparison: the top row compares the simulated displacement with the noisy data while the bottom row shows the restoring force vs. displacement (hysteresis loops) curves for all three models.

4.3.3. *Bayesian model selection: the* `BayesModelSelection` *class*

`UQpy` also supports a Bayesian approach to model selection via its `BayesModelSelection` class. This class is structured in a similar fashion as the `InfoModelSelection` class, i.e., it leverages the `BayesParameterEstimation` class to perform Bayesian parameter estimation for all candidate models. Results of the Bayesian parameter estimation are then used to compute the model evidence $p(\mathrm{D}|m_i)$ for all candidate models m_i, along with the models' posterior probability $P(m_i|\mathrm{D}) \propto p(\mathrm{D}|m_i)P(m_i)$. However, careful consideration must be given to this evidence computation. Currently `UQpy` only supports computation of the model evidence via the harmonic mean method [66], which is known to yield evidence estimates with large variance. This computation will be improved upon in future releases of `UQpy`.

5. Probability of failure and rare-event analysis: the `Reliability` module

Reliability refers to a system's probability of satisfying its intended performance measures under a variety of uncertainties. For structural reliability, the system corresponds to a structure under e.g. material, environmental and loading uncertainty. Mathematically it is defined as the complement of the probability of failure P_f. In its simplest form, the probability of failure P_f is calculated through the performance function $g(\mathrm{X})$, given the uncertain parameters X, as

$$P_f = \mathbb{P}(g(\mathrm{X}) \leq 0) = \int_{\{g(\mathrm{X})\leq 0\}} p(\mathrm{x})d\mathrm{x}, \tag{9}$$

where $\mathbb{P}[\cdot]$ is the probability measure and $p(\mathrm{x})$ is the joint pdf of the parameters X. `UQpy` provides different methods for approximating the integral in Eq. (9) and estimating the reliability of a system. Here, we specifically discuss two approaches that are implemented in the `Reliability` module. The first approach, implemented in the `TaylorSeries` class, is based on a Taylor series expansion of the limit surface, $g(\mathrm{X}) = 0$ and includes two methods: the first order reliability method (FORM) and the second order reliability methods (SORM) [67–71]. The second approach is the simulation-based subset simulation method [47] implemented in the `SubsetSimulation` class. Note that other reliability analysis methods are available in `UQpy`, but these are not exclusive to the `Reliability` module. Various classes in the `SampleMethods` class can be used for Monte Carlo simulation-based reliability analysis, including for example the `IS` class for importance sampling. Additionally, surrogate-model based approaches such as the `AKMCS` class for adaptive Kriging in the `SampleMethods` module can be used to estimate P_f. This is discussed further in Section 6.2.

The `Reliability` module is illustrated using the structural fire example described in Section 2.3. Again, the fire load density is modeled as a uniformly distributed random variable X_1 on the range 50–450 MJ/m^2 and the yield strength at room temperature is modeled as a normally distributed random variable X_2 with a mean value of 250 MPa and a coefficient of variation of 7%, (standard deviation = 17.5 MPa). Failure occurs when the deflection at the midpoint of the beam exceeds the maximum allowable deflection, $P_f = \mathbb{P}(g(\mathrm{X}) \leq 0)$, i.e., the model computes the performance function $g(\mathrm{X}) = d_m(X_1, X_2) - d_a$, where $\mathrm{X} = [X_1, X_2]$ is the vector of the two uncorrelated random variables. The reference solution for the probability of failure is calculated using MCS (see Section 3.1.1) to be $P_f = 3.3\%$. Note that a high probability of failure is selected for ease of illustration. The reliability analysis methods illustrated here are capable of solving problems with much smaller probability of failure.

5.1. *Expansion-based reliability analysis: the* `TaylorSeries` *class*

The `TaylorSeries` class is used to approximate the performance function $g(\mathrm{U})$ through its Taylor series expansion, where $\mathrm{U} \sim \mathcal{N}(\mathbf{0}, \mathrm{I})$, locally at the design point $\mathrm{U}^\star$ defined as the point of maximum probability along the limit surface $g(\mathrm{U}) = 0$. More specifically, the base `TaylorSeries` class possesses two sub-classes, `FORM` and `SORM`, that construct first-order and second-order expansions respectively. For brevity, we discuss only FORM and note that the application of SORM follows directly. In FORM, the performance function is approximated by

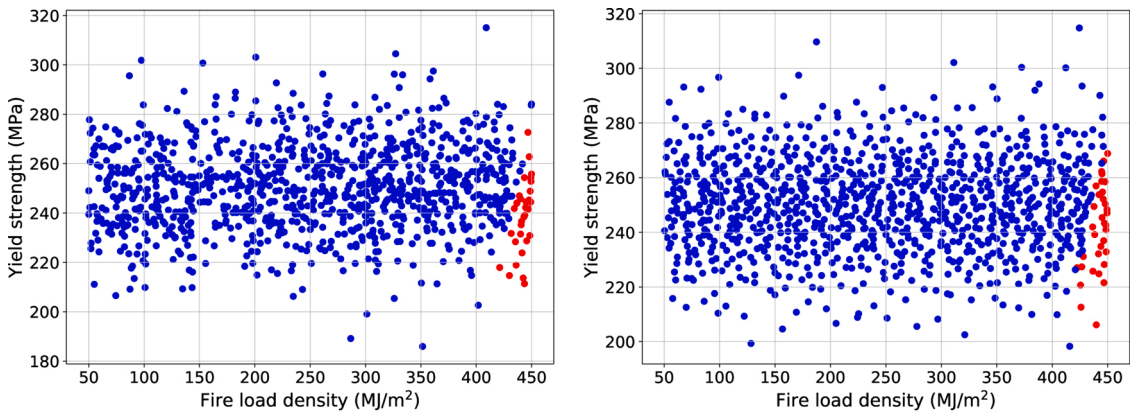
$$g(\mathrm{U}) \approx g(\mathrm{U}^\star) + \nabla g(\mathrm{U}^\star)(\mathrm{U} - \mathrm{U}^\star)^\top, \tag{10}$$

where $\nabla g(\mathrm{U}^\star)$ is the gradient of $g(\mathrm{U})$ evaluated at $\mathrm{U}^\star$. The probability failure is given by $P_{f,\mathrm{form}} = \Phi(-\beta_{\mathrm{HL}})$, where $\Phi(\cdot)$ is the standard normal cumulative distribution function and $\beta_{\mathrm{HL}} = ||\mathbf{U}^\star||$ is the norm of the design point known as the Hasofer-Lind reliability index [72,69,71].

The FORM is assumed to operate on standard normal random variables, which means that a nonlinear iso-probabilistic transformation from the physical variables $\mathrm{X} \sim p(\mathrm{x})$ to uncorrelated standard normal random variables $\mathrm{U} \sim \mathcal{N}(\mathbf{0}, \mathrm{I})$ is required. In `UQpy` this transformation is performed with the `Nataf` class in the `Transformations` module (for details see Appendix D). Moreover, the evaluation of the necessary gradients of the model (`RunModel` object) are performed using a central finite difference approximation.

A `FORM` object contains the the probability distribution models of the random parameters as objects of the `Distribution` class and the computational model as an object of the `RunModel` class, and is instantiated as follows:

```
from UQpy.Reliability import FORM
# Instantiating FORM
Q = FORM(dist_object=dists, runmodel_object=
    abaqus_sfe_model)
```



(a) MCS (b) STS

Fig. 5. 1024 samples generated using (a) the `MCS` class and (b) `STS` class. For samples shown in blue, maximum deflection at the midspan of the beam does not exceed the deflection tolerance of 7.14 cm. For samples shown in red, maximum midspan deflection exceeds the deflection tolerance. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

The specific definitions of `dists` and `abaqus_sfe_model` are presented in Sections 3.1.1 and 2.3, respectively. After instantiating the `FORM` class, the `run` method is called to execute FORM starting at a specified seed point in the parameter space (`seed_x`) or in the uncorrelated standard normal space (`seed_u`). If a seed is not provided, the algorithm will automatically start from the origin in the standard normal space.

```
# Run FORM
Q.run(seed_u=np.array([1, 1])
```

The design point for this problem was estimated to be $\mathrm{U}^{\star} = (1.6, -0.6)$, corresponding to point $\mathrm{X}^{\star} = (429.68, 2.37e8)$ in the parameter space. The design point and the corresponding approximate limit surface are shown in Fig. 13. The probability of failure with FORM was found to be $P_f = 4.2\%$ which is close to the probability of failure estimated from Monte Carlo simulation (3.3%).

5.2. *Simulation-based reliability analysis: the* `SubsetSimulation` *class*

Subset simulation [47], is a simulation-based reliability analysis method that efficiently estimates small failure probabilities by expressing them as a product of larger, intermediate conditional probabilities. That is, the probability of failure P_f is expressed as:

$$P_f = P(F_1)\prod_{i=1}^{m-1} P(F_{i+1}|F_i), \tag{11}$$

where m is the number of conditional levels and, generally the probability of each conditional level $P(F_{i+1}|F_i)$ is reasonably large (i.e. $O(10^{-1})$) such that it can be statistically estimated by performing Monte Carlo simulations with a relatively small number of samples. The Monte Carlo simulations on each conditional level are conducted using various MCMC algorithms that condition on the samples lying in each conditional region, studies of which can be found in the recent literature, e.g. [73,74]. In `UQpy`, the `SubsetSimulation` class can employ any of the MCMC algorithms, in-built or custom, that are available as child classes of the `MCMC` class. This is achieved by directly passing the class and its relevant inputs into the `SubsetSimulation` object.

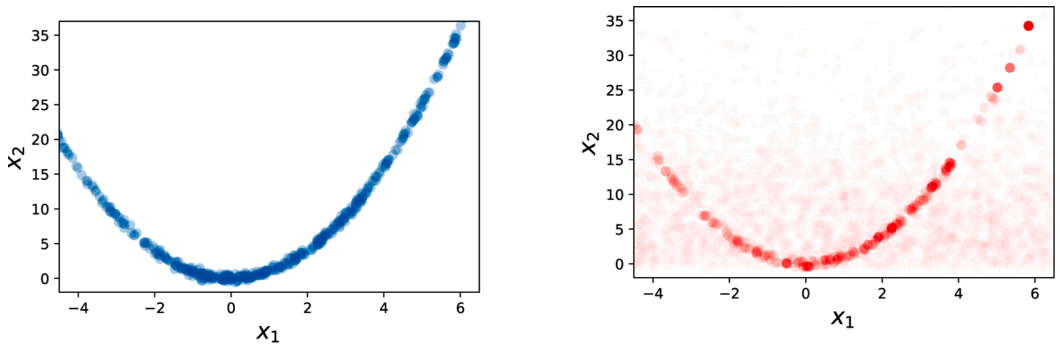
To run subset simulation, it is necessary to define the model object using the `RunModel` class as before, and also to define the `Distribution` object for the probability distribution of the input parameters. Again, using the structural fire example, these are given as follows:

```
abaqus_sfe_model = RunModel(model_script='
    abaqus_subset_sfe_model_script.py',
    input_template='abaqus_input_subset_sfe.py',
    output_script='
    extract_abaqus_output_subset_sfe.py',
    var_names=['qtd', 'fy'], ...)
dist = MVNormal(mean=np.zeros(2), cov=np.eye(2))
```

Because subset simulation is traditionally performed using uncorrelated standard normal random variables, the above `RunModel` object has been defined to take, as input, transformed standard normal random variables. Note that this is for convenience in illustration only and is not required. Subset simulation is then executed using the `MMH` class as follows:

```
from UQpy.Reliability import SubsetSimulation
from UQpy.SampleMethods import MMH
x_ss = SubsetSimulation(mcmc_class=MMH,
    runmodel_object=abaqus_sfe_model,
    p_cond=0.1, nsamples_ss=1000, log_pdf_target=
    dist.log_pdf, dimension=2, nchains=100, ...)
```

and the probability of failure is obtained from the attribute `x_ss.pf`. One execution of this code yields an estimated probability of failure of 3.34%. In this example, the algorithm is designed such that each conditional probability $P(F_i) = 0.1$. Therefore, the algorithm converges using only two levels with the samples shown in Fig. 14.



(a) 500 MCMC samples

(b) 4000 weighted samples from IS - opacity is proportional to the sample weight

Fig. 6. Using the MCMC and IS classes to draw samples from the Rosenbrock distribution.

6. Surrogate modeling and active learning

The construction of fast-running surrogate models (aka meta-models, emulators, or response surfaces) is an important component of UQ. It enables the rapid approximation of model outputs at new parameter values for which a typically expensive, high fidelity physics-based calculation has not been performed. This facilitates UQ for both forward and inverse problems. Additionally, surrogate models are important for active machine learning for uncertainty analyses.

Recognizing the importance of surrogate models, the `Surrogates` class of `UQpy` currently has two classes that enable surrogate model construction. The first is the stochastic reduced order model (SROM), implemented in the `SROM` class. SROMs, developed by Grigoriu [75], approximate the distribution of the output of a stochastic model by optimally fitting sample weights to the model evaluations. For brevity, the `SROM` class is not discussed further here. The second is the `Kriging` class, which implements Gaussian process regression or Kriging surrogates. In the sections below, we discuss the construction of Kriging models with the `Kriging` class and their use in active learning, specifically for adaptive Kriging with various learning functions and adaptive Monte Carlo analyses.

6.1. *Gaussian process regression/kriging: the* `Kriging` *class*

Kriging is an interpolation technique in which the interpolant is assumed to be the sum of a regression model and a realization of Gaussian random process as

$$\widehat{y}(x) = \mathcal{F}(x;\beta) + Z(x). \tag{12}$$

The regression model, $\mathcal{F}$, is a linear combination of p chosen basis functions having parameters $\beta_i, i = 1,\ldots,p$. The Gaussian random process, $Z(x)$, has mean zero and covariance defined through a correlation matrix, $\mathcal{R}(x^{(i)},x^{(j)};\theta)$, having hyperparameters θ estimated from a set of sample training points. That is:

$$\begin{aligned}\mathcal{F}(x;\quad \beta) &= \beta_1 f_1(x) + \cdots + \beta_p f_p(x) = f(x)^T\beta,\\ E[Z(x^{(i)})Z(x^{(j)})] &= \sigma^2\mathcal{R}(x^{(i)},x^{(j)};\theta).\end{aligned}$$

The regression coefficients and Gaussian process variance are estimated by solving a generalized-least square problem ($C^{-1}(Y - F\beta) = 0$, where C^{-1} is the Cholesky decomposition of R) such that,

$$\beta^* = \left(F^TR^{-1}F\right)^{-1}F^TR^{-1}Y,$$

$$\sigma^{*2} = \frac{1}{N}(Y - F\beta^*)^TR^{-1}\left(Y - F\beta^*\right),$$

where F is the matrix of basis function evaluations at the training points, R is the correlation matrix also evaluated at the training points, and Y are the training data (i.e. model evaluations). For a detailed description refer to [76]. The Kriging hyperparameters are estimated by solving the maximum likelihood problem. Since the output is assumed to follow a multivariate Gaussian distribution ($Y \sim \mathcal{N}(F\beta,\ \sigma^2\mathcal{R})$), the marginal log-likelihood can be defined as,

$$\log(p(Y|\beta^*,\sigma^*,\theta)) = -\frac{1}{2}\log(|R|) - \frac{N}{2}log(2\pi\sigma^{*2}) - \frac{N}{2}. \tag{13}$$

Once the hyperparameters are known, the regression coefficients and process variance are updated and the Best Linear Unbiased Predictor (BLUP) of the model can be computed at new sample points as:

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

and the variance of the estimator is computed as:

$$\sigma^2_{\widehat{y}(x)} = \sigma^2(1 - r(x)^TR^{-1}r(x) + u(x)^T(F^TR^{-1}F)^{-1}u(x)),$$

where $r(x)$ is the correlation matrix between the new prediction point x and the existing training points and $u(x) = F^TR^{-1}r(x) - f(x)$.

To employ the `Kriging` class, the user first needs to define regression and correlation models to initiate the object. All correlation models in `UQpy` are assumed to take a product form as:

$$R_{ij} = \mathcal{R}(x^{(i)},x^{(j)};\theta) = \prod_{k=1}^{d}\mathcal{R}_k(x_k^{(i)} - x_k^{(j)};\theta).$$

Multiple in-built regression and correlation models are available as listed in Table 1.

The `Kriging` class also allows for user-defined regression and correlation functions. These can be passed to the `reg_model` and `corr_model` arguments as callable functions, making extension of the `Kriging` class straightforward. The user is referred to the `UQpy` documentation [32] for a detailed explanation.

Here, we define a `Kriging` object that will be employed throughout this section as follows:

```
from UQpy.Surrogates import Kriging
metamodel = Kriging(reg_model='Linear',
    corr_model='Exponential', nopt=1,
    corr_model_params=[1, 1], random_state=2)
```

where `nopt` controls the number of times the maximum likelihood problem is solved with different random starting points when the training is performed.

Next, the `Kriging` model must be trained using the available data.

Let $\{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), \ldots, (\mathbf{x}^{(N)}, \mathbf{y}^{(N)})\}$ be the training data, where $\mathbf{x}^{(i)} \in \mathbb{R}^d$ and $\mathbf{y}^{(i)} \in \mathbb{R}^q$ for all $i \in \{1, \ldots, N\}$. In this step, the `Kriging.fit` method is used to solve the maximum likelihood problem (Eq. (13)) to estimate the hyperparameters, θ, and the subsequent least squares fit for β and σ. These parameters are saved in the `corr_model_params` attribute and can be updated by subsequent calls to the `fit` method as additional data are provided. Given training data, `X` and `Y`, provided as `numpy` arrays or lists, the training is performed as follows:

```
metamodel.fit(samples=X, values=Y)
```

After training the surrogate model, the object can be used to approximate the model at new, untrained sample points. The `Kriging` class object has two methods for prediction. `Kriging.predict` returns the prediction and its standard deviation at specified sample points and `Kriging.jacobian` returns the gradient of the model at specified sample points. This is illustrated as follows:

```
y, y_sd = metamodel.predict(x=new_samples),
    return_std=True)
grad = metamodel.jacobian(x=new_samples)
```

6.2. Adaptive kriging: the `AKMCS` class

The previous section introduces the user to the basics of constructing a `Kriging` object, training it, and using it for prediction. Here, we illustrate how it can be used adaptively for various active learning problems. For these adaptive Kriging models, we generally adopt the terminology Adaptive Kriging with Monte Carlo simulation (AKMCS) from [77] although the methods included here are referred to in the literature under different names depending on the learning function that is employed. In the `SampleMethods` module, the `AKMCS` class provides five in-built learning functions, shown in Table 2, to tackle problems ranging from reliability analysis, to optimization and global fit. The `AKMCS` class also accepts user-defined learning functions, thus providing an easy way to develop new methods within this active learning context.

The workflow of the `AKMCS` class is straightforward. To initialize it, the user provides an initial set of samples, a `Distribution` object to generate learning set of samples, the `RunModel` object for model execution, a `Kriging` object, and sets the relevant parameters. For example, to initialize `AKMCS` for reliability analysis of the structural fire finite element model in Abaqus, the commands are as follows:

```
from UQpy.SampleMethods import RectangularStrata,
    RectangularSTS, AKMCS
strata = RectangularStrata(nstrata=[5, 5])
x_sts = RectangularSTS(dist_object=dists,
    strata_object=strata, nsamples_per_stratum=1,
    random_state=1)
akmcs = AKMCS(samples=x_sts.samples,
    run_model_object=abaqus_sfe_model,
    krig_object=metamodel, nlearn=10^5,
    learning_function='U', dist_object=dists,
    random_state=3)
```

Upon initialization, the `AKMCS` class will execute the model at the provided sample points and train the `Kriging` object. In each iteration, the `LHS` class is used to generate `nlearn` random samples at which the learning function (`learning_function`) is evaluated. Based on the learning function evaluations, new samples are selected for model evaluation using the specified `RunModel` object. This learning process is carried out by calling the `run` method as follows:

```
akmcs.run(nsamples=100)
```

where `nsamples` is the final total number of model evaluations (not the number of additional model evaluations).

Fig. 15(a) illustrates the samples generated using the `AKMCS` class for the structural fire reliability problem. The initial `STS` samples are shown in black and the `AKMCS` samples are shown in blue. Circles (solid and hollow) are used for samples with maximum displacement within the tolerance, i.e. safe samples, and x's are used for samples whose displacement exceeds the tolerance, i.e. in the failure region. Also, the safe region (as defined by the Kriging model) is shaded with green and the failure region is shaded with red. As expected from the 'U' learning function, samples are generated close to the limit surface separating the failure and safe domains. Fig. 15(b) shows the probability of failure estimate after each sample is added compared with the probability of failure estimate from 10,000 MCS (i.e. 3.3%) samples.

As previously mentioned, the `AKMCS` class allows the user to define their own learning function, thus allowing straightforward extensions of the existing framework. This architecture is used in various places withing the `UQpy` code, such as within the `SampleMethods.LHS` class to define novel sampling criteria, or within the `Inference.InferenceModel` class to utilize user-defined likelihood functions. This architecture can be illustrated schematically in the context of the `AKMCS` class as follows:

```
# Existing AKMCS framework
class AKMCS:
    def __init__(self, learning_function, ...):
        # initialization of class attributes
        self.learning_function =
            learning_function
    def run(nsamples, ...):
        if self.learning_function in ['U',
            'Weighted-U', 'EFF', 'EIF', 'EIGF']:
            # run AK-MCS design based on
            # supported learning functions
            self.samples = ...
        else:
            # run AK-MCS design based on
            # custom learning function
            self.samples = ...

# Define custom learning function and integrate
# within existing framework
def custom_learning_function(surr, pop, ...):
    # compute selected samples from pop and
    # indicator for stopping criterion based
    # on surrogate model surr
    return selected_pop, learning_fun_value,
        indicator_stop
akmcs = AKMCS(learning_function=
    custom_learning_function, nsamples=10, ...)
```

Here, the user only needs to write the function associated with their custom learning function to accept a surrogate model (having a `.predict` method), a population of points at which to evaluate the learning function and other neccesary parameters. No intrusion with the `AKMCS` class is necessary.

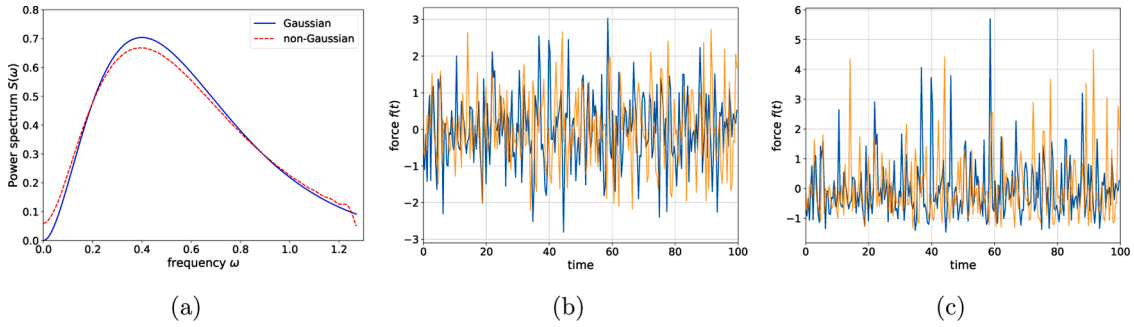


Fig. 7. (a) Gaussian and non-Gaussian power spectra. (b) Sample realizations of a Gaussian stochastic processes simulated using the `SRM` class. (c) Sample realizations of a lognormal translation process simulated using the `SRM` and `Translation` classes.

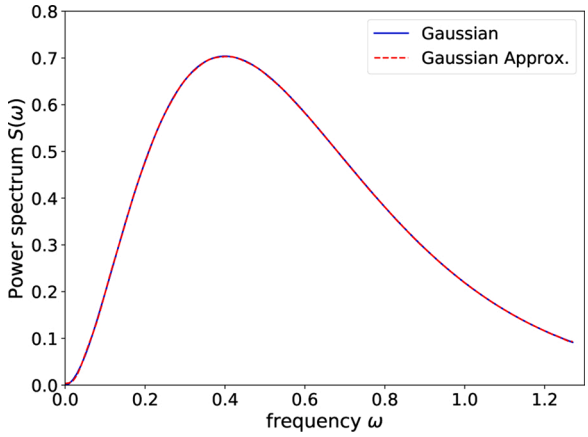


Fig. 8. True Gaussian power spectrum and the Gaussian power spectrum identified from the lognormal power spectrum using the `Inverse-Translation` class.

6.3. *Refined stratified sampling: the* `RSS` *class*

Refined stratified sampling (RSS) [43] and gradient enhanced refined stratified sampling (GE-RSS) [44] build upon stratified sampling designs to generate new samples either randomly (RSS) or based on variation in the model output (GE-RSS). In each of these methods, strata are selected from an existing stratified design and divided according to a specified refinement criterion. A new sample is then drawn randomly within the new stratum. See [43,44] for algorithmic details.

In the `SampleMethods` module, the `RSS` class can execute either RSS or GE-RSS to extend the design. The `RSS` class has two child classes (i.e. `RectangularRSS` and `VoronoiRSS`), which generates samples in rectangular and voronoi stratification. Here, we illustrate the GE-RSS method as an active learning method for voronoi stratification that leverages gradients estimated using the `Kriging` class. Because `VoronoiRSS` operates on stratified designs, the user must first create an initial `VoronoiSTS` object. If the user specifies only the `VoronoiSTS` object, then `VoronoiRSS` is initialized. If the user also specifies a `RunModel` object and a `Kriging` object, then GE-RSS is employed as follows:

```
from  UQpy.SampleMethods import VoronoiStrata,
    VoronoiSTS, VoronoiRSS
strata = VoronoiStrata(nseeds=15, dimension=2)
x_sts = VoronoiSTS(dist_object=dists,
    strata_object=strata, nsamples_per_stratum=1,
     random_state=1)
rss = VoronoiRSS(sample_object=x_sts,
    run_model_object=abaqus_sfe_model,
    krig_object=metamodel, random_state=3)
```

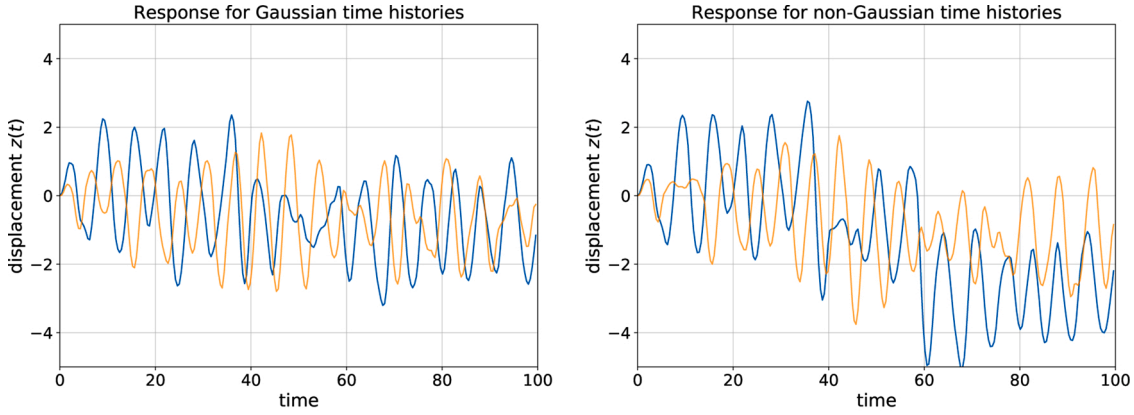


Fig. 9. Two realizations of the response of the SDOF Bouc Wen dynamics model to (a) Gaussian and (b) lognormal excitation.

Once the RSS class has been initiated, the user can extend the sample design by invoking the run method and providing the total number of samples (not the number of samples to add) as follows:

```
rss.run(nsamples=100)
```

An example of samples generated using the RSS class to execute 100 samples using GE-RSS with Voronoi stratification and a Kriging surrogate model for gradient estimation for the structural fire finite element model is shown in Fig. 16. Fig. 16(a) shows a 3-dimensional plot of the Kriging surrogate, along with the STS samples (black dots) and RSS

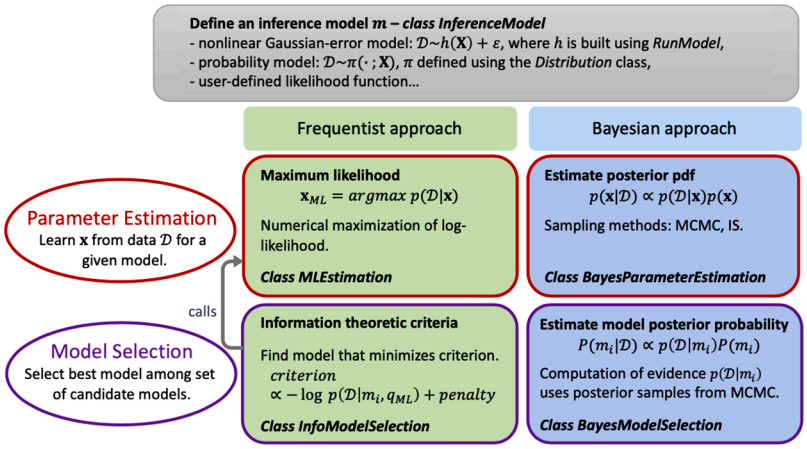


Fig. 10. Overview of the Inference module.

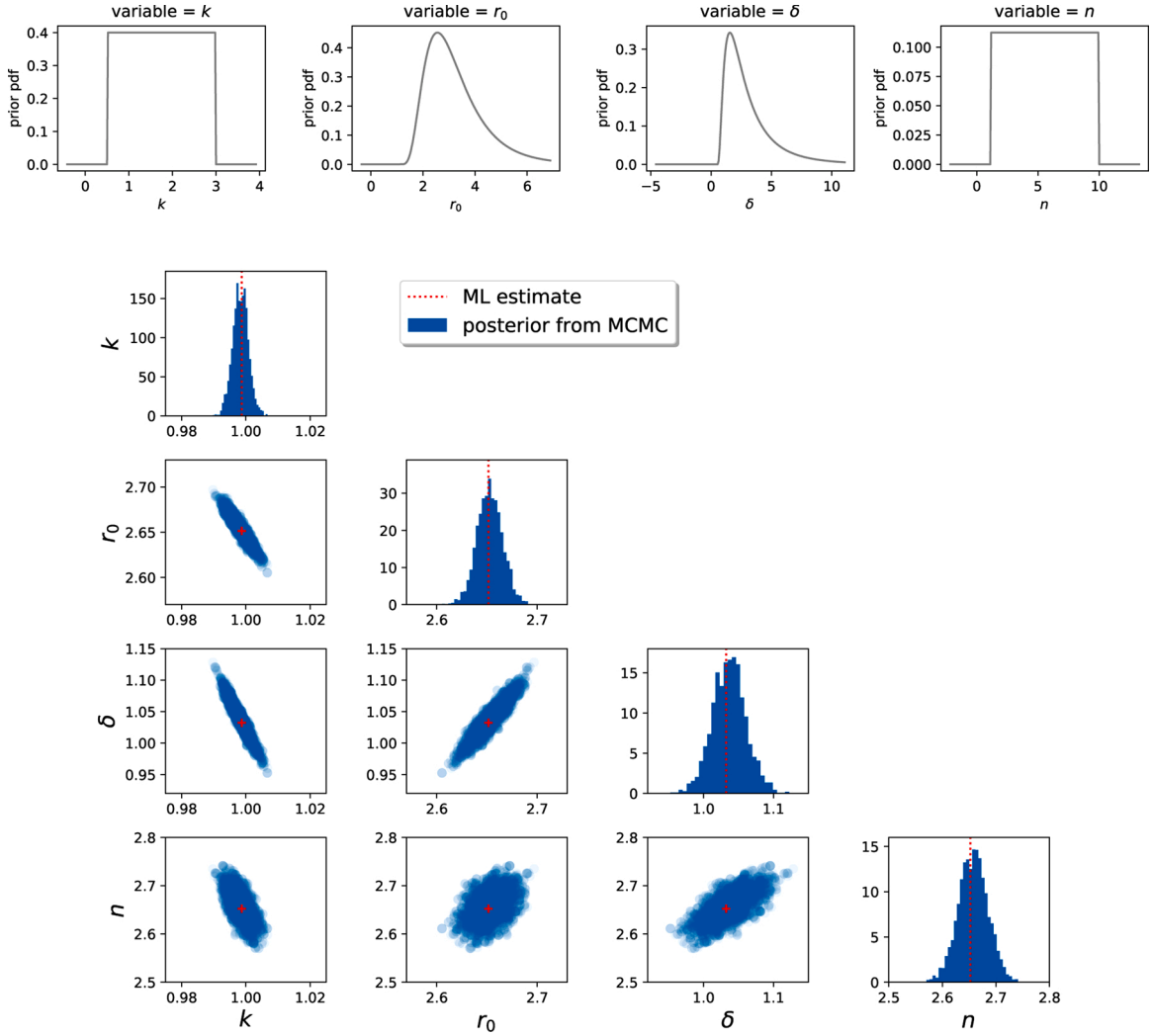


Fig. 11. Illustration of Bayesian parameter estimation for the Bouc-Wen model. Top: parameter prior probability densities (having independent marginals) for Bayesian analysis. Bottom: results of the parameter estimation for the Bouc-Wen hysteresis model from noisy displacement data.

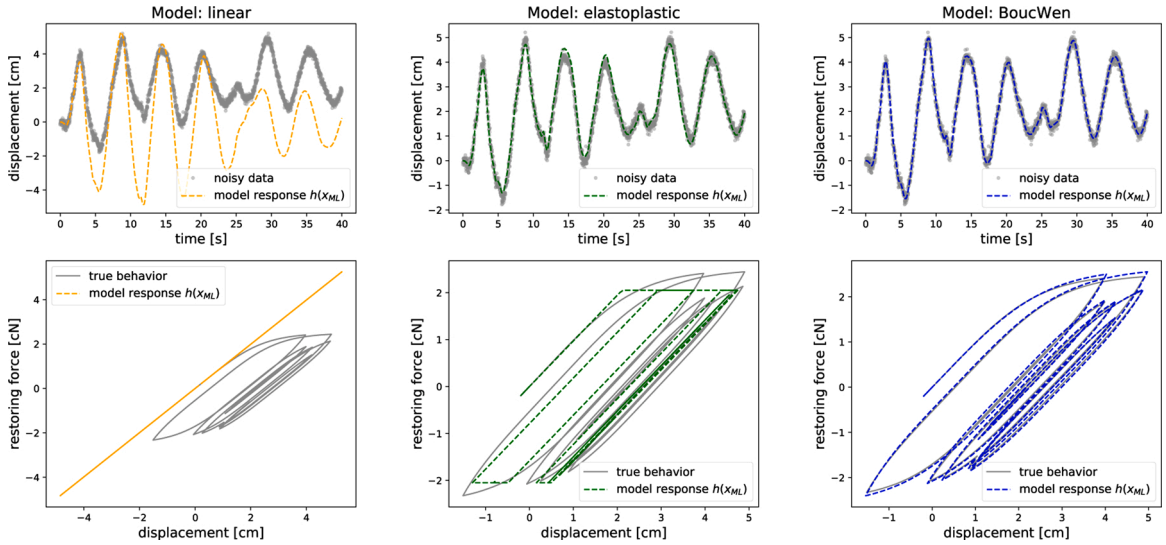


Fig. 12. Comparison of three SDOF dynamical models estimated from noisy displacement data, the model responses are obtained by running a simulation with the ML parameter estimate.

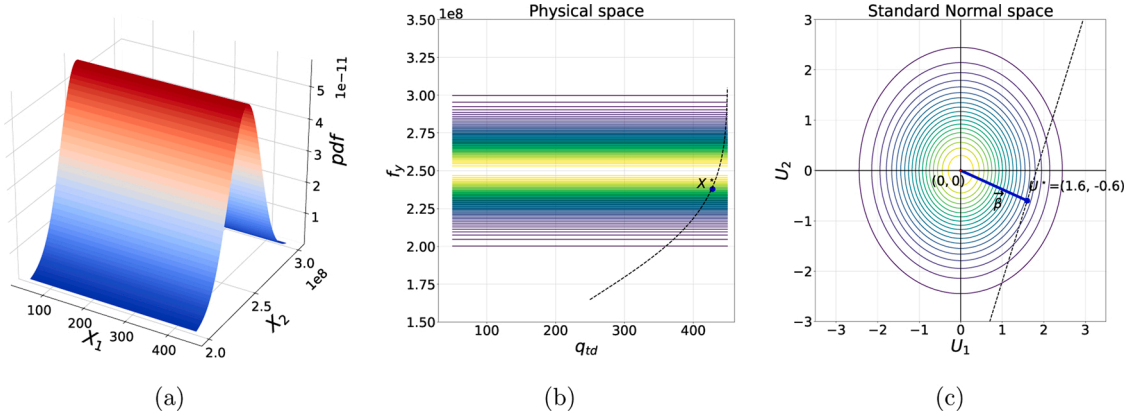


Fig. 13. (a) Joint probability density function for the two random variables. Illustration of the design point and the FORM limit surface in (b) the physical parameter space and (c) the standard normal space.

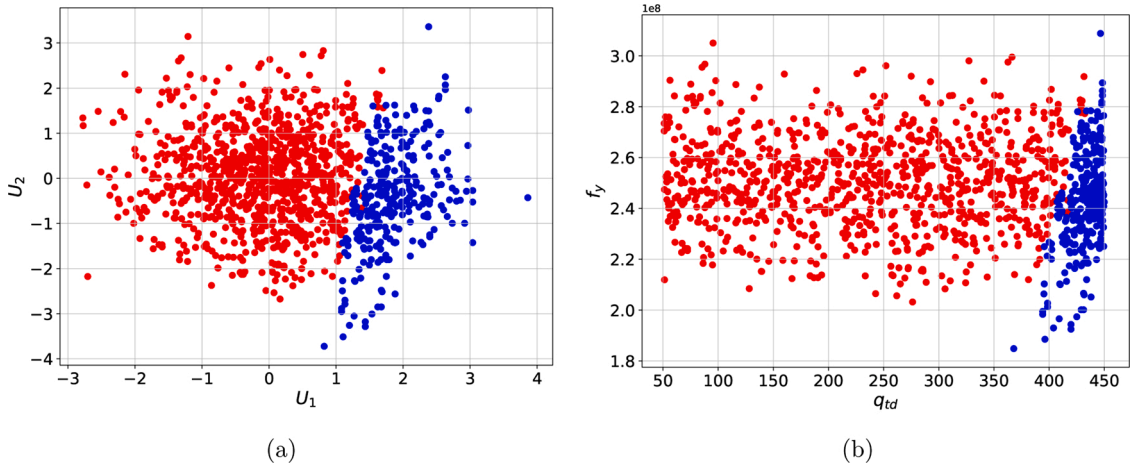


Fig. 14. Samples drawn using the `SubsetSimulation` class in `UQpy` with 1000 samples in each level. Samples are shown in (a) the parameter space, and (b) the standard normal space. The red markers indicate samples in the first level and blue markers show the samples in the second level. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 1
Built-in correlation and regression models for the `Kriging` class. Note: $\delta_k = x_k^{(i)} - x_k^{(j)}$, $\zeta_k = \min\{1, \theta_k|\delta_k|\}$ for cubic and spherical correlation functions and $\phi_k = \theta_k|\delta_k|$ for spline correlation functions.

Correlation model (`corr_model`)		Regression model (`reg_model`)	
Name	$\mathcal{R}_k(\theta, \delta_k)$	Name	$F(x) = [f_1(x), f_2(x), \ldots]$
`'Exponential'`	$\exp(-\theta_k\|\delta_k\|)$	`'Constant'`	$f_1(x) = 1$
`'Gaussian'`	$\exp(-\theta_k\delta_k^2)$		
`'Linear'`	$\max\{0, 1-\theta_k\|\delta_k\|\}$	`'Linear'`	$f_1(x) = 1, f_2(x) = x_1, \ldots, f_{d+1}(x) = x_d$
`'Cubic'`	$1 - 3\zeta_k^2 + 2\zeta_k^3$		
`'Spherical'`	$1 - 1.5\zeta_k + 0.5\zeta_k^3$		
`'Spline'`	$\begin{cases} 1 - 15\phi_k^2 + 30{*}\phi_k^3 & \text{for } 0 \le \phi_k \le 0.2 \\ 1.25(1-\phi_k)^3 & \text{for } 0.2 \le \phi_k \le 1 \\ 0 & \text{for } \phi_k \ge 1 \end{cases}$	`'Quadratic'`	$f_1(x) = 1, f_2(x) = x_1, f_3(x) = x_2, \ldots, f_{d+1}(x) = x_d, f_{d+2}(x) = x_1^2, f_{d+3}(x) = x_1x_2, \ldots, f_{2d+1}(x) = x_1x_d, f_{2d+2}(x) = x_2^2, f_{2d+3}(x) = x_2x_3, \ldots, f_{3d}(x) = x_2x_d, \ldots, f_{\frac{(d+1)(d+2)}{2}} = x_d^2$

Table 2
Existing `AKMCS` learning functions. Notes on notation: $\widehat{y}(x)$ = Kriging surrogate model; $\sigma_{\widehat{y}}(x)$ = Kriging standard deviation; $\Phi(\cdot)$ = Standard Normal CDF; $\phi(x)$ = Standard Normal PDF; f_{min} = Current minimum value; x^* = closest point to the present sample.

Learning function	Argument	Objective	Equation
U-function [77]	`'U'`	Reliability	$U(x) = \frac{\|\widehat{y}(x)\|}{\sigma_{\widehat{y}}(x)}$
Weighted U-function [78]	`'Weighted-U'`	Reliability	$w(x) = \frac{max_x[p(x)] - p(x)}{max_x[p(x)]} U(x)$
Expected Feasibility Function [79]	`'EFF'`	Reliability	$EFF(x) = \widehat{y}(x)[2\Phi(\frac{-\widehat{y}(x)}{\sigma_{\widehat{y}}(x)}) - \Phi(\frac{-\sigma_{\widehat{y}}(x) - \widehat{y}(x)}{\sigma_{\widehat{y}}(x)}) - \Phi(\frac{\sigma_{\widehat{y}}(x) - \widehat{y}(x)}{\sigma_{\widehat{y}}(x)})] - \sigma_{\widehat{y}}(x)[2\phi(\frac{-\widehat{y}(x)}{\sigma_{\widehat{y}}(x)}) - \phi(\frac{-\sigma_{\widehat{y}}(x) - \widehat{y}(x)}{\sigma_{\widehat{y}}(x)}) - \phi(\frac{\sigma_{\widehat{y}}(x) - \widehat{y}(x)}{\sigma_{\widehat{y}}(x)})] + \sigma_{\widehat{y}}(x)[\Phi(\frac{-\sigma_{\widehat{y}}(x)\widehat{y}(x)}{\sigma_{\widehat{y}}(x)}) - \Phi(\frac{\sigma_{\widehat{y}}(x) - \widehat{y}(x)}{\sigma_{\widehat{y}}(x)})]$
Expected Improvement Function [80]	`'EIF'`	Optimization	$EIF(x) = (f_{min} - \widehat{y}(x))\Phi(\frac{f_{min} - \widehat{y}(x)}{\sigma_{\widehat{y}}(x)}) + \sigma_{\widehat{y}}(x)\phi(\frac{f_{min} - \widehat{y}(x)}{\sigma_{\widehat{y}}(x)})$
Expected Improvement for Global Fit [81]	`'EIGF'`	Global Fit	$EIGF(x) = (\widehat{y}(x) - y(x^*))^2 + \sigma_{\widehat{y}}^2(x)$

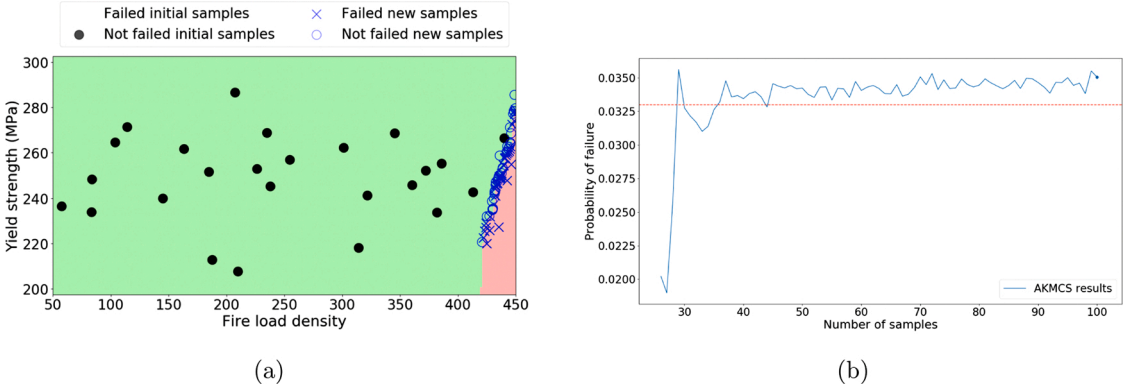


Fig. 15. (a) `AKMCS` samples delineating the failure and safe domains; and (b) probability of failure at each iteration of the `AKMCS` class.

samples (green dots). Fig. 16(b) illustrate the Voronoi stratification of parameter space (black boundaries represent Voronoi cells) and the corresponding Delaunay triangulation (orange boundaries). Notice that the GE-RSS method focuses samples in the region of the parameter space with high variation in model output.

7. Concluding remarks

This paper presented the `UQpy` software toolbox for uncertainty quantification (UQ) in Python. The software serves as both a user-ready toolbox that includes many of the latest methods for UQ in

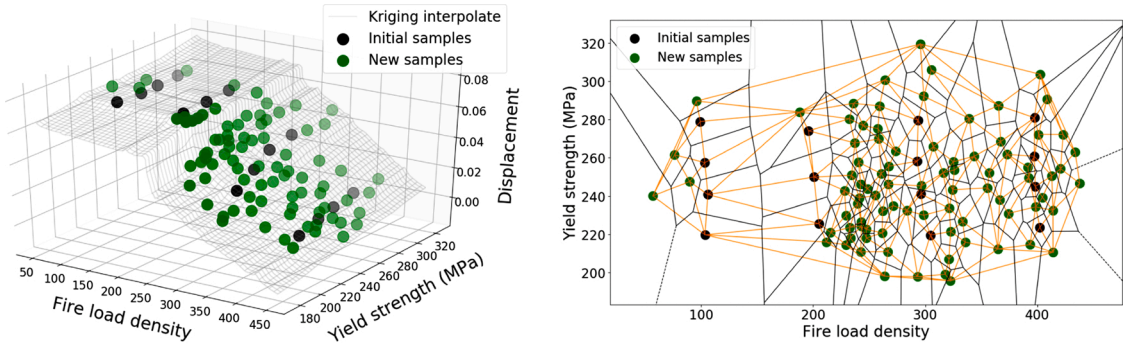


Fig. 16. (a) 3D scatter plot of the `RSS` samples with their model evaluations and the Kriging surrogate model, (b) Voronoi stratification (black) and Delaunay triangulation (orange) of the parameter space. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 3
Current `UQpy` capabilities organized by Module and Class structure.

Module	Class	Description	Introduced
`RunModel`	`RunModel`	Execute computational model	1.0.0
`Distributions`	See Appendix C	Define distribution objects in `UQpy`	2.0.0
`SampleMethods`	`AKMCS`	Adaptive Kriging with Monte Carlo Simulation	3.0.0
	`IS`	Importance Sampling	1.3.0
	`LHS`	Latin Hypercube Sampling	1.1.0
	`MCMC, MH, MMH, DRAM, DREAM, Stretch`	Markov Chain Monte Carlo	1.1.0
	`MCS`	Monte Carlo Sampling	1.1.0
	`RSS`	Refined Stratified Sampling	2.0.0
	`Simplex`	Uniform Sampling over a simplex element	2.0.0
	`Strata`	Defines a Strata object for STS/RSS	1.0.0
	`STS`	Stratified Sampling	1.1.0
`Transformations`	`Correlate`	Induces correlation in standard normal samples	1.0.0
	`Decorrelate`	Removes correlation in standard normal samples.	1.0.0
	`Nataf`	Transform non-Gaussian samples to standard normal	1.0.0
		Transform standard normal samples to a prescribed distribution	
		Calculates the distortion of the non-Gaussian correlation matrix	
		Calculates the distortion of the Gaussian correlation matrix	
`Surrogates`	`Kriging`	Gaussian Process Regression (Kriging)	2.0.0
	`SROM`	Stochastic Reduced Order Model	1.0.0
`Reliability`	`SubsetSimulation`	Subset Simulation	1.0.0
	`TaylorSeries, FORM, SORM`	First Order Reliability Method (FORM)	2.0.0
		Second Order Reliability Method (SORM)	
`Inference`	`BayesModelSelection`	Bayesian Model Selection	2.0.0
	`BayesParameterEstimation`	Bayesian Parameter Estimation	2.0.0
	`InferenceModel`	Model Definition for Inference	2.0.0
	`InfoModelSelection`	Information Theoretic Model Selection (AIC/BIC)	2.0.0
	`MLEstimation`	Maximum Likelihood Parameter Estimation	2.0.0
`StochasticProcess`	`BSRM`	Bispectral Representation Method	2.0.0
	`InverseTranslation`	Iterative Translation Approximation Method	2.0.0
	`KLE`	Karhunen-Loéve Expansion	2.0.0
	`SRM`	Spectral Representation Method	2.0.0
	`Translation`	Translation Process	2.0.0
`Utilities`	None	A collection of methods used across modules	2.0.0

computational modeling and a convenient development environment for Python programmers advancing the field of UQ. The paper presents an introduction to the software's existing capabilities in forward propagation of uncertainties and sampling methods, generation of random processes and random fields, probabilistic inverse modeling including parameter estimation and model selection, reliability analysis, surrogate modeling, and active learning. The paper is not intended to be a comprehensive "deep-dive" into the software. For this, the reader is referred to the `UQpy` documentation [32]. Instead it is intended to highlight the structure of the code, many of its capabilities, its

Table 4
Available distributions in `UQpy`.

Available distributions in `UQpy`		
Subclasses of `DistributionContinuous1D`		
Distribution	Class name	Parameters and default values
Beta	`Beta`	`[a, b, loc=0., scale=1.]` `a, b` $\in \mathbb{R}_{>0}$
Cauchy	`Cauchy`	`loc=0., scale=1.]`
Chi-Squared	`ChiSquare`	`df, loc=0., scale=1.`
Exponential	`Exponential`	`loc=0., scale=1.` common parameterization with `lambda=1/scale`
Gamma	`Gamma`	`a, loc=0., scale=1.` `a` $\in \mathbb{R}_{>0}$
Generalized Extreme Value	`GenExtreme`	`c, loc=0., scale=1.` `c` $\in \mathbb{R}_{\geq 0}$
Inverse Gaussian	`InvGauss`	`mu, loc=0, scale=1]` `mu` $\in \mathbb{R}_{>0}$
Laplace	`Laplace`	`loc=0., scale=1.`
Levy	`Levy`	`loc=0., scale=1.`
Logistic	`Logistic`	`loc=0., scale-1.`
Lognormal	`Lognormal`	`s, loc=0., scale=1.` common parameterization `s`=σ, `scale`=$exp(\mu)$
Maxwell-Boltzmann	`Maxwell`	`loc=0., scale=1.`
Normal (Gaussian)	`Normal`	`loc=0., scale=1.` `loc`=μ, `scale`=σ
Pareto	`Pareto`	`b, loc=0., scale=1.` `b` $\in \mathbb{R}_{\geq 0}$
Rayleigh	`Rayleigh`	`loc=0., scale=1.`
Truncated Normal	`TruncNorm`	`a, b, loc=0., scale=1.` `a`=$\left(\frac{cliplow-\mu}{\sigma}\right)$, `b`=$\left(\frac{cliphigh-\mu}{\sigma}\right)$ `loc`=μ, `scale`=σ
Uniform	`Uniform`	`loc=0., scale=1.` lower and upper bounds are `[loc, loc+scale]`
Subclasses of `DistributionDiscrete1D`		
Binomial	`Binomial`	`n, p, loc=0.` `n` $\in \mathbb{N}_0$, `p` $\in [0,1]$
Poisson	`Poisson`	`mu, loc=0.`
Subclasses of `DistributionND`		
Multivariate Normal	`MVNormal`	`mean, cov=1.`
Multinomial	`Multinomial`	`n, p` `n` $\in \mathbb{N}_0$, `p[i]` $\in [0,1]$ and $\sum$`p[i]`=1

applications in computational modeling, and most importantly its capacity to serve as a platform for UQ methodology development in Python. Emphasis is placed on the structure/architecture of the code as this provides the reader with valuable insights into how to develop new methodology within the code. In particular, the paper highlights the `RunModel` model, which serves as a generic interface to models of all kinds and is used to drive simulations and uncertainty analyses performed in `UQpy`. To illustrate the various capabilities, two examples are tracked throughout the paper and analyzed repeatedly using different methods. The first is a Python model solving a nonlinear structural dynamics problem using explicit time integration. The second is a third-party Abaqus model solving the thermomechanical response of a beam structure.

The developments presented herein relate specifically to `UQpy` Version 3, which is available for download from GitHub [10]. All the scripts running the various examples are provided as supplementary materials and can also be downloaded from GitHub [11].

Author contributions

Audrey Olivier: Conceptualization, methodology, software, validation, data curation, writing – original draft, visualization. Dimitris G. Giovanis: Conceptualization, methodology, software, validation, writing – review & editing, visualization. B.S. Aakash: Methodology, software, validation, writing – review & editing, visualization. Mohit Chauhan: Methodology, software, validation, writing – review & editing, visualization. Lohit Vandanapu: Methodology, software, validation, writing – review & editing, visualization. Michael D. Shields: Conceptualization, methodology, software, resources, writing – review & editing, supervision, project administration, funding acquisition.

Conflict of interest

The authors declare that there is no conflict of interest.

Declaration of Competing Interest

The authors report no declarations of interest.

Acknowledgements

Development of the `UQpy` software has been indirectly supported by several agencies that have generously supported work related to its core functionalities including the Office of Naval Research (Award numbers

N00014-15-1-2754, N00014-16-1-2582, and N00014-18-1-2644), National Science Foundation (Award number 1652044), Army Research Laboratory (Award number W911NF-12-2-0022), and the Department of Energy (Award number DE-SC0020428). Also, part of the examples presented in the manuscript were executed using computational resources at the Maryland Advanced Research Computing Center (MARCC).

Appendix A. UQpy modules and classes

A complete list of modules and classes available in Version 3.0 is provided in Table 3.

Appendix B. UQpy extensions in progress

UQpy is a rapidly evolving code and new components are being added continually. The following modules are currently under development, but are not ready for release. Note, however, that some source code can be found on the open UQpy Github repository.

- `DimensionReduction`: Perform linear or nonlinear dimension reduction for high-dimensional problems. Will be released with Version 3.
- `Sensitivity`: Perform global and local sensitivity analysis.
- `Collocation`: Stochastic collocation methods.

Additionally, within the existing and new modules a number of new classes and methods are under development or are anticipated in the near future, including the following:

- `Surrogates.PCE`: Polynomial chaos expansion based surrogate models.
- `DimensionReduction.Grassmann`: Grassmann manifold projection-based dimension reduction.
- `DimensionReduction.DiffusionMaps`: Nonlinear dimension reduction with diffusion maps.
- `DimensionReduction.LinearBasis`: Linear dimension reduction of a high-dimensional array using SVD or HO-SVD.
- `Sensitivity.Morris`: Computation of sensitivity indices via the method of Morris.
- `Sensitivity.Sobol`: Estimation of Sobol sensitivity indices.
- `Collocation.SparseGrid`: Sparse-grid stochastic collocation.
- `Collocation.MultiElement`: Multi-element stochastic collocation.
- `Collocation.Simplex`: Simplex stochastic collocation.
- `SampleMethods.QMC`: Quasi-Monte Carlo sampling.
- `SampleMethods.PSS`: Partially stratified sampling.
- `SampleMethods.LSS`: Latinized stratified sampling.
- `SampleMethods.SparseGrid`: Sparse-grid structured points for numerical integration.

These new modules and classes will further enable the implementation and development of more advanced algorithms such as adaptive stochastic collocation methods, dimension-reduction, and Monte Carlo sampling. Please note that the list above is subject to change in future releases.

Appendix C. Defining probability distribution objects in UQpy: the `Distributions` module

Being a largely probabilistic code, many tasks in `UQpy` rely on probability distributions. The `Distributions` module is used to define probability distribution objects. These objects possess various methods that allow the user to: compute the probability density function (`pdf` method), cumulative distribution function (`cdf`), the logarithm of the pdf (`log_pdf`), return the moments (`moments`), draw independent samples (`rvs`) and fit the parameters of the model from data (`fit`).

The `Distributions` module is built upon four base classes that are used to construct specific distributions via subclassing, thus allowing the user to easily build custom distributions that can be integrated within the existing UQpy framework. The `Distribution` class is the parent class to all distribution classes, `DistributionContinuous1D` and `DistributionDiscrete1D` are the base classes for univariate continuous and discrete distributions respectively, while the `DistributionND` class is the base class for multivariate distributions. These base classes cannot be used directly as a distribution, instead they define certain methods that are common to all distributions, such as the `get_params` and `update_params` methods that allow the user to return/update the parameters of a distribution for instance. A specific distribution is created via subclassing of the base classes – `UQpy` implements a number of well-known distributions as shown in Table 4, with parameters that adhere to those defined in the `scipy.stats` package. In order to instantiate a univariate normal distribution object for instance, the commands are as follows:

```
from UQpy.Distributions import Normal
p1 = Normal(loc=0., scale=2.)
```

UQpy also allows to create multivariate distributions from its marginals, potentially adding dependence via a copula, through its classes `JointInd` and `JointCopula`. Both these classes are subclasses of the `DistributionND` base class, and a user could easily create other custom classes that define distributions as a combination (sum, product etc.) of existing distributions. The following code instantiates a bi-variate distribution object with standard normal marginals and Gumbel copula dependence:

```
from UQpy.Distributions import Normal, Gumbel, JointCopula
p2 = JointCopula(marginals=[Normal(), Normal()], copula=Gumbel(theta=2.))
```

Its methods are called as follows:

```
pdf_values = p2.pdf(x=x)
```

In the statement above, the input `x` must be a 2D ndarray of shape (N, d), where d is the dimension of the distribution and N is the number of points at which to evaluate the pdf. A detailed description of the `Distribution` methods and their parameters can be found in `UQpy`'s documentation [32].

As previously mentioned, custom distributions can be easily built via direct subclassing of the appropriate base classes. For example, the user can define the distribution `Rosenbrock` with the `pdf` and `log_pdf` methods as follows:

```
from UQpy.Distributions import DistributionND
class Rosenbrock(DistributionND):
    def __init__(self, p=20.):
        super().__init__(p=p)
    def pdf(self, x):
        return np.exp(-(100*(x[:, 1]-x[:, 0]**2)**2+(1-x[:, 0])**2)/self.params['p
    ''])
    def log_pdf(self, x):
        return -(100*(x[:, 1]-x[:,0]**2)**2+(1-x[:,0])**2/self.params['p']
```

The custom Rosenbrock `Distribution` object is then instantiated as follows:

```
p3 = Rosenbrock(p=20)
```

and the `pdf` method can be invoked as follows:

```
pdf3 = p3.pdf(x=x)
```

Appendix D. Isoprobabilistic transformations: the `Transformations` module

`UQpy` includes widely used isoprobabilistic transformations, most notably the Nataf transformation [82] to transform arbitrarily distributed random variables to standard normal variables. That is, given a random vector x having marginal cdfs $F_i(x_i)$, the Nataf transformation can be employed to map to a correlated standard normal random vector z. Similarly, the inverse Nataf transformation can be used to map from z→x. Through a linear transformation, z can then be mapped to an uncorrelated standard normal random vector U, and vice versa. This is illustrated as follows:

$$\mathbf{x} \sim \left(F_i\left(x_i\right)\Big|_{i=1,\ldots,n},\ \mathbf{R} = \left[\xi_{\mathrm{ij}}\right]\right) \underset{\text{Inv. Nataf}}{\overset{\text{Nataf}}{\Longleftrightarrow}} \mathrm{z} \sim \mathrm{N}\left(0, \mathbf{R}_0 = \left[\rho_{\mathrm{ij}}\right]\right) \underset{\text{Correlate}}{\overset{\text{Decorrelate}}{\Longleftrightarrow}} \mathrm{u} \sim \mathrm{N}\left(0, \mathrm{I}\right) \tag{14}$$

The Nataf transformation is very useful when conducting probabilistic modeling, for example when performing reliability analysis using first and second order reliability methods (FORM/SORM). The mapping of the i^{th} component of $\mathbf{x}$ to the normal space $\mathbf{z}$ is achieved through the transformation $z_i = \Phi^{-1}(F_i(x_i))$, where $\Phi(\cdot)$ is the standard normal cumulative distribution function. The mapping from z→x results in a correlation distortion that can be solved through following integral:

$$\xi_{\mathrm{ij}} = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}\left(\frac{x_i-\mu_i}{\sigma_i}\right)\left(\frac{x_j-\mu_j}{\sigma_j}\right)\varphi_2(z_i, z_j, \rho_{\mathrm{ij}})\mathrm{d}z_i\mathrm{d}z_j, \tag{15}$$

where μ_i and σ_i are the mean and standard deviation of random variable x_i, respectively and $\varphi_2(\cdot,\cdot,\rho)$ is the bivariate standard normal probability density function with correlation coefficient ρ_{ij} [57]. `UQpy` computes this integral numerically using a standard quadratic two-dimensional Gauss-Legendre integration scheme. However, the inverse expression, i.e. identifying the Gaussian correlation ρ_{ij} from a known non-Gaussian correlation ξ_{ij} in the mapping from x→z is not defined in closed-form. This inverse correlation distortion therefore requires the use of an iterative procedure. The `Nataf` class utilizes the Iterative Translation Approximation Method (ITAM) [58]. This method identifies an underlying correlated Gaussian random vector that, when mapped to the non-Gaussian distribution produces a non-Gaussian correlation that is as close as possible to the prescribed value considering the potential for Nataf incompatibility [58]. Several methods are available in the `Nataf` class of the `Transformations` module to perform these transformations. A `Nataf` object for a two-dimensional random vector x with non-Gaussian marginal distributions defined by `dist1` and `dist2`, is instantiated as:

```
from UQpy.Transformations import Nataf
#Instantiate a Nataf transformation object
nataf_obj = Nataf(dist_object=[dist1,dist2], corr_x=Rx)
#or
nataf_obj = Nataf(dist_object=[dist1,dist2], corr_z=Rz)
```

The returned `nataf_obj` object computes the distorted correlation matrix in the standard normal space `corr_z` (estimated with the `distortion_x2z` method with parameters `beta`, `itam_error1` and `itam_error2` that are specific to the ITAM method – see documentation [32]), if `corr_x` is given, or the distorted correlation matrix in the parameter space `corr_x` (estimated with the `distortion_z2x` method) if `corr_z` is given. After instantiating the `Nataf` object we can sample from the joint pdf of the random vector **x** with the method `rvs` as:

```
x = nataf_obj.rvs(nsamples)
```

where `nsamples` is the number of samples to be drawn and then, transform the set `x` to standard normal samples using the method `transform_x2z` as:

```
z = nataf_obj.transform_x2z(x, jacobian=True)
```

where `jacobian` is the Jacobian of the transformation (returned if `True`). Finally, the inverse Nataf transformation, which is widely used in reliability analysis using FORM, can be performed with the method `transform_z2x` as:

```
x = nataf_obj.transform_z2x(z, jacobian=True)
```

in order to transform a set `z` of correlated standard normal samples to non-Gaussian samples `x`.

The `Transformations` module also allows to induce or remove correlation from a standard normal vector with the classes `Correlate` and `Decorrelate`, respectively. A set of uncorrelated normal variables `u` can be made to possess correlation `Rz` as follows:

```
from UQpy.Transformations import Correlate
z = Correlate(u, Rz).samples_z
```

The correlated standard normal random vector **z** is obtained from **u** as $z = H_0 u$, where H_0 is the lower-triangular Cholesky decomposition of matrix $R_z = [\rho_{ij}]$, such that $H_0 H_0^{\top} = R_0$. Similarly, correlation can be removed as follows:

```
from UQpy.Transformations import Decorrelate
u = Decorrelate(z, Rz).samples_u
```

The uncorrelated standard normal random vector **u** is obtained from **z** as $u = H_0^{-1} z$.

Appendix E. Supplementary data

Supplementary material related to this article can be found, in the online version, at doi:https://doi.org/10.1016/j.jocs.2020.101204.

References

[1] B. Sudret, Global sensitivity analysis using polynomial chaos expansions, Reliab. Eng. Syst. Saf. 93 (2008) 964–979, https://doi.org/10.1016/j.ress.2007.04.002.

[2] B. Sudret, Meta-models for structural reliability and uncertainty quantification, in: Asian-Pacific Symposium on Structural Reliability and Its Applications, Singapore, 2012, pp. 1–24.

[3] A. Nikishova, A.G. Hoekstra, Semi-intrusive uncertainty propagation for multiscale models, J. Comput. Sci. 35 (2019) 80–90, https://doi.org/10.1016/j.jocs.2019.06.007.

[4] J. Sturdy, J.K. Kjernlie, H.M. Nydal, V.G. Eck, L.R. Hellevik, Uncertainty quantification of computational coronary stenosis assessment and model based mitigation of image resolution limitations, J. Comput. Sci. 31 (2019) 137–150, https://doi.org/10.1016/j.jocs.2019.01.004.

[5] R. Archibald, M. Chakoumakos, T. Zhuang, Characterizing the elements of earth's radiative budget: applying uncertainty quantification to the CESM, J. Comput. Sci. 5 (2014) 85–89, https://doi.org/10.1016/j.jocs.2013.03.001.

[6] D.M. Tartakovsky, Assessment and management of risk in subsurface hydrology: a review and perspective, Adv. Water Resour. 51 (2013) 247–260, https://doi.org/10.1016/j.advwatres.2012.04.007.

[7] J.A. Vrugt, C.J.F. ter Braak, M.P. Clark, J.M. Hyman, B.A. Robinson, Treatment of input uncertainty in hydrologic modeling: doing hydrology backward with Markov chain Monte Carlo simulation, Adv. Water Resour. 51 (2013) 247–260, https://doi.org/10.1029/2007WR006720.

[8] N. Linde, D. Ginsbourger, J. Irving, F. Nobile, A. Doucet, On uncertainty quantification in hydrogeology and hydrogeophysics, Adv. Water Resour. 110 (2017) 166–181, https://doi.org/10.1016/j.advwatres.2017.10.014.

[9] A. Rafiei Emam, M. Kappas, S. Fassnacht, N.H.K. Linh, Uncertainty analysis of hydrological modeling in a tropical area using different algorithms, Front. Earth Sci. 12 (2018) 661–671, https://doi.org/10.1007/s11707-018-0695-y.

[10] Shields Uncertainty Research Group, UQpy – uncertainty quantification with python. https://github.com/SURGroup/UQpy. (Accessed 24 July 2020).

[11] Shields Uncertainty Research Group, Jupyter example scripts, supplementary materials to UQpy manuscript. https://github.com/SURGroup/UQpy_paper. (Accessed 24 July 2020).

[12] G.I. Schuëller, H.J. Pradlwarter, Computational stochastic structural analysis (COSSAN) – a software tool, Struct. Saf. 28 (1–2) (2006) 68–82, https://doi.org/10.1016/j.strusafe.2005.03.005.

[13] S. Marelli, B. Sudret, UQLab: a framework for uncertainty quantification in Matlab. Vulnerability, Uncertainty, and Risk: Quantification, Mitigation, and Management, 2014, pp. 2554–2563.

[14] Software – Engineering Risk Analysis Group – Technical University of Munich. https://www.bgu.tum.de/era/software/. (Accessed 11 June 2020).

[15] D. Gorissen, I. Couckuyt, P. Demeester, T. Dhaene, K. Crombecq, A surrogate modeling and adaptive sampling toolbox for computer based design, J. Mach. Learn. Res. 11 (2010) 2051–2055.

[16] J. Bourinet, C. Mattrand, V. Dubourg, A review of recent features and improvements added to FERUM software, Proc. of the 10th International Conference on Structural Safety and Reliability (ICOSSAR'09) (2009).

[17] D. Dupuy, C. Helbert, J. Franco, DiceDesign and DiceEval: two R packages for design and analysis of computer experiments, J. Stat. Softw. 65 (11) (2015) 1–38, https://doi.org/10.18637/jss.v065.i11.

[18] O. Roustant, D. Ginsbourger, Y. Deville, DiceKriging, DiceOptim: two R packages for the analysis of computer experiments by kriging-based metamodeling and optimization, J. Stat. Softw. 51 (1) (2012) 1–55, https://doi.org/10.18637/jss.v051.i01.

[19] C. Walter, G. Defaux, B. Iooss, V. Mouroussamy, Package 'mistral', 2014.

[20] B. Iooss, A. Janon, G. Pujol, Package 'sensitivity', 2015.

[21] B.M. Adams, W. Bohnhoff, K. Dalbey, J. Eddy, M. Eldred, D. Gay, K. Haskell, P. D. Hough, L.P. Swiler, DAKOTA, a Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis: Version 5.0 User's Manual, Sandia National Laboratories, Tech. Rep. SAND2010-2183, 2009.

[22] B. Debusschere, K. Sargsyan, C. Safta, K. Chowdhary, Uncertainty quantification toolkit (UQTk). Handbook of Uncertainty Quantification, 2016, pp. 1–21.

[23] M. Baudin, A. Dutfoy, B. Iooss, A.-L. Popelin, OpenTURNS: an industrial software for uncertainty quantification in simulation. Handbook of Uncertainty Quantification, 2017, pp. 2001–2038.

[24] B.H. Thacker, D.S. Riha, S.H. Fitch, L.J. Huyse, J.B. Pleming, Probabilistic engineering analysis using the NESSUS software, Struct. Saf. 28 (1–2) (2006) 83–107, https://doi.org/10.1016/j.strusafe.2004.11.003.

[25] S. Tennøe, G. Halnes, G.T. Einevoll, Uncertainpy: a python toolbox for uncertainty quantification and sensitivity analysis in computational neuroscience, Front. Neuroinform. 12 (2018) 49, https://doi.org/10.3389/fninf.2018.00049.

[26] V. Puzyrev, M. Ghommem, S. Meka, pyROM: a computational framework for reduced order modeling, J. Comput. Sci. 30 (2019) 157–173, https://doi.org/10.1016/j.jocs.2018.12.004.

[27] J. Feinberg, H.P. Langtangen, Chaospy: an open source tool for designing methods of uncertainty quantification, J. Comput. Sci. 11 (2015) 46–57, https://doi.org/10.1016/j.jocs.2015.08.008.

[28] J. Sukys, M. Kattwinkel, SPUX: Scalable Particle Markov Chain Monte Carlo for Uncertainty Quantification in Stochastic Ecological Models, 2018, pp. 159–168, https://doi.org/10.3233/978-1-61499-843-3-159.

[29] R. Dutta, M. Schoengens, L. Pacchiardi, A. Ummadisingu, N. Widmer, P. Kunzli, J.-P. Onnela, A. Mira, ABCpy: An High-Performance Computing Perspective to Approximate Bayesian Computation, 2020 arXiv:1711.04694.

[30] S.M. Martin, D. Walchli, G. Arampatzis, P. Koumoutsakos, Korali: A High-Performance Computing Framework for Stochastic Optimization and Bayesian Uncertainty Quantification, 2020 arXiv:2005.13457.

[31] C. Wang, Q. Duan, C.H. Tong, Z. Di, W. Gong, A GUI platform for uncertainty quantification of complex dynamical models, Environ. Model. Softw. 76 (2016) 1–12, https://doi.org/10.1016/j.envsoft.2015.11.004.

[32] Shields Uncertainty Research Group, Johns Hopkins University, UQpy Documentation. https://uqpyproject.readthedocs.io/. (Accessed 11 June 2020).

[33] R. Cimrman, V. Lukeš, E. Rohan, Multiscale finite element calculations in Python using SfePy, Adv. Comput. Math. (2019), https://doi.org/10.1007/s10444-019-09666-0.

[34] O. Tange, GNU Parallel 2018, 2018, https://doi.org/10.5281/zenodo.1146014.

[35] M. Ismail, F. Ikhouane, J. Rodellar, The hysteresis Bouc-Wen model, a survey, Arch. Comput. Methods Eng. 16 (2) (2009) 161–188, https://doi.org/10.1007/s11831-009-9031-8.

[36] Center for Engineering Strong Motion Data, El Centro Earthquake of 18 May 1940, 2017. . (Accessed 29 May 2020), https://strongmotioncenter.org/.

[37] ABAQUS, ABAQUS Documentation, Dassault Systèmes Simulia Corp, United States, 2019.

[38] M. Gillie, Analysis of heated structures: nature and modelling benchmarks, Fire Saf. J. 44 (5) (2009) 673–680, https://doi.org/10.1016/j.firesaf.2009.01.003.

[39] CEN, EN 1991-1-2: Eurocode 1: Actions on Structures – Part 1–2: General Actions – Actions on Structures Exposed to Fire, 2002.

[40] CEN, EN 1363-1: Fire Resistance Tests. Part 1: General Requirements, 2012.

[41] M.D. McKay, R.J. Beckman, W.J. Conover, Comparison of three methods for selecting values of input variables in the analysis of output from a computer code, Technometrics 21 (2) (1979) 239–245, https://doi.org/10.1080/00401706.1979.10489755.

[42] M.D. Shields, J. Zhang, The generalization of Latin hypercube sampling, Reliab. Eng. Syst. Saf. 148 (2016) 96–108, https://doi.org/10.1016/j.ress.2015.12.002.

[43] M.D. Shields, K. Teferra, A. Hapij, R.P. Daddazio, Refined stratified sampling for efficient Monte Carlo based uncertainty quantification, Reliab. Eng. Syst. Saf. 142 (2015) 310–325, https://doi.org/10.1016/j.ress.2015.05.023.

[44] M.D. Shields, Adaptive Monte Carlo analysis for strongly nonlinear stochastic systems, Reliab. Eng. Syst. Saf. 175 (2018) 207–224, https://doi.org/10.1016/j.ress.2018.03.018.

[45] R.C. Smith, Uncertainty Quantification: Theory, Implementation, and Applications, Society for Industrial and Applied Mathematics, 2014.

[46] A. Gelman, H.S. Stern, J.B. Carlin, D.B. Dunson, A. Vehtari, D.B. Rubin, Bayesian Data Analysis, Chapman and Hall/CRC, 2013.

[47] S.-K. Au, J.L. Beck, Estimation of small failure probabilities in high dimensions by subset simulation, Probab. Eng. Mech. 16 (4) (2001) 263–277, https://doi.org/10.1016/S0266-8920(01)00019-4.

[48] H. Haario, M. Laine, A. Mira, E. Saksman, DRAM: efficient adaptive MCMC, Stat. Comput. 16 (4) (2006) 339–354, https://doi.org/10.1007/s11222-006-9438-0.

[49] J. Vrugt, C. Braak, C. Diks, B. Robinson, J. Hyman, D. Higdon, Accelerating Markov chain Monte Carlo simulation by differential evolution with self-adaptive randomized subspace sampling, Int. J. Nonlinear Sci. Numer. Simul. 10 (3) (2009) 273–290.

[50] J.A. Vrugt, Markov chain Monte Carlo simulation using the DREAM software package: theory, concepts, and MATLAB implementation, Environ. Model. Softw. 75 (2016) 273–316, https://doi.org/10.1016/j.envsoft.2015.08.013.

[51] J. Goodman, J. Weare, Ensemble samplers with affine invariance, Commun. Appl. Math. Comput. Sci. 5 (1) (2010) 65–80, https://doi.org/10.2140/camcos.2010.5.65.

[52] D. Foreman-Mackey, D.W. Hogg, D. Lang, J. Goodman, emcee: The MCMC Hammer, Publ. Astron. Soc. Pac. 125 (925) (2013) 306–312, https://doi.org/10.1086/670067.

[53] M. Shinozuka, C.-M. Jan, Digital simulation of random processes and its applications, J. Sound Vib. 25 (1) (1972) 111–128, https://doi.org/10.1016/0022-460X(72)90600-1.

[54] M. Shinozuka, G. Deodatis, Simulation of stochastic processes by spectral representation, Appl. Mech. Rev. 44 (4) (1991) 191–204, https://doi.org/10.1115/1.3119501.

[55] K. Phoon, S. Huang, S. Quek, Simulation of second-order processes using Karhunen-Loeve expansion, Comput. Struct. 80 (12) (2002) 1049–1060, https://doi.org/10.1016/S0045-7949(02)00064-0.

[56] W. Betz, I. Papaioannou, D. Straub, Numerical methods for the discretization of random fields by means of the Karhunen-Loève expansion, Comput. Methods Appl. Mech. Eng. 271 (2014) 109–129, https://doi.org/10.1016/j.cma.2013.12.010.

[57] M. Grigoriu, Applied Non-Gaussian Processes: Examples, Theory, Simulation, Linear Random Vibration, and MATLAB Solutions, Prentice Hall, 1995.

[58] M. Shields, G. Deodatis, P. Bocchini, A simple and efficient methodology to approximate a general non-Gaussian stationary stochastic process by a translation process, Probab. Eng. Mech. 26 (4) (2011) 511–519, https://doi.org/10.1016/j.probengmech.2011.04.003.

[59] M.D. Shields, H. Kim, Simulation of higher-order stochastic processes by spectral representation, Probab. Eng. Mech. 47 (2017) 1–15, https://doi.org/10.1016/j.probengmech.2016.11.001.

[60] B.A. Benowitz, G. Deodatis, Simulation of wind velocities on long span structures: a novel stochastic wave based model, J. Wind Eng. Ind. Aerodyn. 147 (2015) 154–163, https://doi.org/10.1016/j.jweia.2015.10.004.

[61] J.L. Beck, K.-V. Yuen, Model selection using response measurements: Bayesian probabilistic approach, J. Eng. Mech. 130 (2) (2004) 192–203, https://doi.org/10.1061/(ASCE)0733-9399(2004)130:2(192.

[62] A.E. Raftery, Bayesian model selection in social research, Sociol. Methodol. 25 (1995) 111–164, https://doi.org/10.2307/271063.

[63] H. Akaike, A new look at the statistical model identification. Selected Papers of Hirotugu Akaike, Springer, 1974, pp. 215–222.

[64] C.M. Hurvich, C.-L. Tsai, Regression and time series model selection in small samples, Biometrika 76 (2) (1989) 297–307, https://doi.org/10.1093/biomet/76.2.297.

[65] C.M. Hurvich, C.-L. Tsai, Model selection for extended quasi-likelihood models in small samples, Biometrics (1995) 1077–1084, https://doi.org/10.2307/2533006.

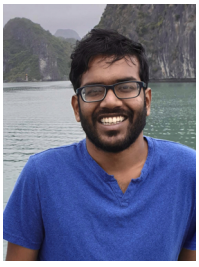
[66] A. Raftery, M. Newton, J. Satagopan, P. Krivitsky, Estimating the integrated likelihood via posterior simulation using the harmonic mean identity, Bayesian Stat. 8 (2007) 1–45.
[67] C. Cornell, A probability-based structural code, J. ACI 66 (1969) 974–985, https://doi.org/10.14359/7446.
[68] O. Ditlevsen, Structural Reliability and the Invariance Problem, Report No. 22, University of Waterloo, Solid Mechanics Division, Waterloo, Canada, 1973.
[69] A.-M. Hasofer, N.-C. Lind, An exact and invariant first-order reliability format, J. Eng. Mech. 100 (1974) 111–121.
[70] O. Ditlevsen, Model uncertainty in structural reliability, Struct. Saf. 1 (1) (1982) 73–86, https://doi.org/10.1016/0167-4730(82)90016-9.
[71] K. Breitung, Asymptotic approximations for probability integrals, Probab. Eng. Mech. 4 (4) (1989) 187–190, https://doi.org/10.1016/0266-8920(89)90024-6.
[72] R. Rackwitz, B. Fiessler, Structural reliability under combined load sequences, Comput. Struct. 9 (5) (1978) 489–494, https://doi.org/10.1016/0045-7949(78)90046-9.
[73] I. Papaioannou, W. Betz, K. Zwirglmaier, D. Straub, MCMC algorithms for subset simulation, Probab. Eng. Mech. 41 (2015) 89–103, https://doi.org/10.1016/j.probengmech.2015.06.006.
[74] M.D. Shields, D. Giovanis, V. Sundar, Subset simulation for problems with strongly non-Gaussian, highly anisotropic, and degenerate distributions, Comput. Struct. (2020) (in review, provisionally accepted).
[75] M. Grigoriu, Reduced order models for random functions. Application to stochastic problems, Appl. Math. Model. 33 (1) (2009) 161–175, https://doi.org/10.1016/j.apm.2007.10.023.
[76] B.W. Santner, T.W. Notz, The Design and Analysis of Computer Experiments, Springer, New York, NY, 2003.
[77] B. Echard, N. Gayton, M. Lemaire, AK-MCS: an active learning reliability method combining kriging and monte carlo simulation, Struct. Saf. 33 (2) (2011) 145–154, https://doi.org/10.1016/j.strusafe.2011.01.002.
[78] V. Sundar, M.D. Shields, Reliability analysis using adaptive kriging surrogates with multimodel inference, ASCE-ASME J. Risk Uncertain. Eng. Syst. A: Civ. Eng. 5 (2) (2019) 04019004, https://doi.org/10.1061/AJRUA6.0001005.
[79] B.J. Bichon, M.S. Eldred, L.P. Swiler, S. Mahadevan, J.M. McFarland, Efficient global reliability analysis for nonlinear implicit performance functions, AIAA J. 46 (10) (2008) 2459–2468, https://doi.org/10.2514/1.34321.
[80] D.R. Jones, M. Schonlau, W.J. Welch, Efficient global optimization of expensive black-box functions, J. Glob. Optim. 13 (4) (1998) 455–492, https://doi.org/10.1023/A:1008306431147.
[81] C.Q. Lam, Sequential Adaptive Designs in Computer Experiments for Response Surface Model Fit (Ph.D. thesis), The Ohio State University, 2008.
[82] R. Lebrun, A. Dutfoy, An innovating analysis of the Nataf transformation from the copula viewpoint, Probab. Eng. Mech. 24 (3) (2009) 312–320, https://doi.org/10.1016/j.probengmech.2008.08.001.

Audrey Olivier holds a Diplôme d'Ingénieur from École Centrale de Nantes, France, and a Ph.D. from the Dept. of Civil Engineering and Engineering Mechanics at Columbia University (2017). She is currently a Postdoctoral Fellow in the Hopkins Extreme Materials Institute at Johns Hopkins University. Dr. Olivier's research interests lie at the junction of physical modeling and data analysis and their application to system monitoring and enhanced computational mechanics. She has worked on the development of efficient inverse uncertainty quantification tools for structural health monitoring applications, or the development of machine learning algorithms for materials modeling.

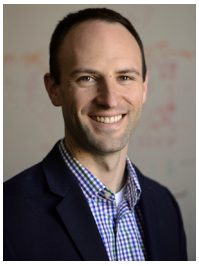
Dimitris Giovanis is an Assistant Research Professor in the Department of Civil and Systems Engineering at Johns Hopkins University. He joined the University in 2016 as a Postdoctoral fellow. He earned his five-year Diploma in Civil Engineering, his M.Sc. in Computational Mechanics from the Department of Chemical Engineering and his Ph.D. in Civil Engineering from the National Technical University of Athens in Greece. His primary research interests are data-driven uncertainty quantification (UQ) approaches for mathematically characterizing parametric and model-form uncertainties, that will inform decision making and eventually lead to the design of high performance physical and structural systems.

B. S. Aakash is a Ph.D. student in the Department of Civil and Systems Engineering at Johns Hopkins University. He holds a Master's degree in Civil Engineering from the Indian Institute of Science, after which he was employed for two years in the Geomechanics Research Laboratory at the Indian Institute of Science. Aakash's research focuses on the development of a Bayesian approach to the treatment of model form uncertainty, with applications in structural fire engineering.

Mohit S. Chauhan is a graduate student in the Dept. of Civil and Systems Engineering at Johns Hopkins University. He joined the Ph.D. program under in 2017. He has completed his undergraduate studies in Civil Engineering and received his Masters in Structural Engineering from Indian Institute of Technology, Kanpur (IITK) in 2017. As a graduate student at IITK, he has worked in the field of Structural Health Monitoring. At Hopkins, he conducts research to develop algorithms for computationally expensive models that learns samples adaptively to tackle problems related to variance reduction and sensitivity analysis.



Lohit Vandanapu is a Graduate Student in the Dept. of Civil and Systems Engineering at Johns Hopkins University. His research primarily focuses on developing frameworks for simulation of higher-order stochastic processes and its applications in various fields of science and engineering. He received his Bachelor of Technology in Civil Engineering from Indian Institute of Technology, Roorkee in 2017.



Michael D. Shields is an Associate Professor in the Dept. of Civil and Systems Engineering at Johns Hopkins University. Prof. Shields conducts methodological research in uncertainty quantification and stochastic simulation for problems in computational mechanics. He received his Ph.D. in Civil Engineering and Engineering Mechanics from Columbia University in 2010 after which he was employed as a Research Engineer in applied computational mechanics at Weidlinger Associates, Inc. He joined the faculty at JHU in 2013. For his work in UQ, Prof. Shields has been awarded the ONR Young Investigator Award, the NSF CAREER Award, and the DOE Early Career Award.