UQLab user manual
The UQLink module

M. Moustapha, S. Marelli, B. Sudret

Chair of Risk, Safety and Uncertainty Quantification
Stefano-Franscini-Platz 5
CH-8093 Zürich

How to cite UQLab

S. Marelli, and B. Sudret, UQLab: A framework for uncertainty quantification in Matlab, Proc. 2nd Int. Conf. on Vulnerability, Risk Analysis and Management (ICVRAM2014), Liverpool, United Kingdom, 2014, 2554-2563.

How to cite this manual

M. Moustapha, S. Marelli and B. Sudret, UQLab user manual – The UQLink module, Report UQLab-V1.2-110, Chair of Risk, Safety & Uncertainty Quantification, ETH Zurich, 2019.

BibTeX entry

```
@TECHREPORT{UQdoc_12_110,
author = {Moustapha, M. and Marelli, S. and Sudret, B.},
title = {{UQLab user manual -- The UQLink module}},
institution = {Chair of Risk, Safety \& Uncertainty Quantification, ETH Zurich},
year = {2019},
note = {Report \# UQLab-V1.2-110}
}
```

Document Data Sheet

Document Ref.	UQLAB-V1.2-110
Title:	UQLAB user manual – The UQLINK module
Authors:	M. Moustapha, S. Marelli, B. Sudret Chair of Risk, Safety and Uncertainty Quantification, ETH Zurich, Switzerland
Date:	22/02/2019

Doc. Version	Date	Comments
V 1.2	22/02/2019	UQLAB V1.2 release
V 1.1	05/07/2018	Initial release

Abstract

Computational models are used nowadays in most fields of natural, social and engineering sciences. The purpose of the UQLab platform is to quantify the impact of uncertainties in the input parameters onto the predictions of such models. Due to their ever increasing complexity, *computational models* are often coded in specialized software. This manual presents a simple way of linking such third-party software to the UQLab platform.
The manual consists of four sections, namely a short description of the concepts of computational model and wrapper, a section on usage comprising commented examples, some applications examples using open-source or commercial finite element software and a reference list.

Keywords: UQLab, UQLink, Wrapper, Third-party software, Computational model

Contents

Chapter 1

Theory

1.1 Introduction

According to the general framework of uncertainty quantification introduced in Sudret (2007); de Rocquigny et al. (2008), a computational model can refer to a physical system, a set of assessment criteria or any other kind of workflow that propagates a set of input parameters to a set of output quantities of interest (Figure 1).

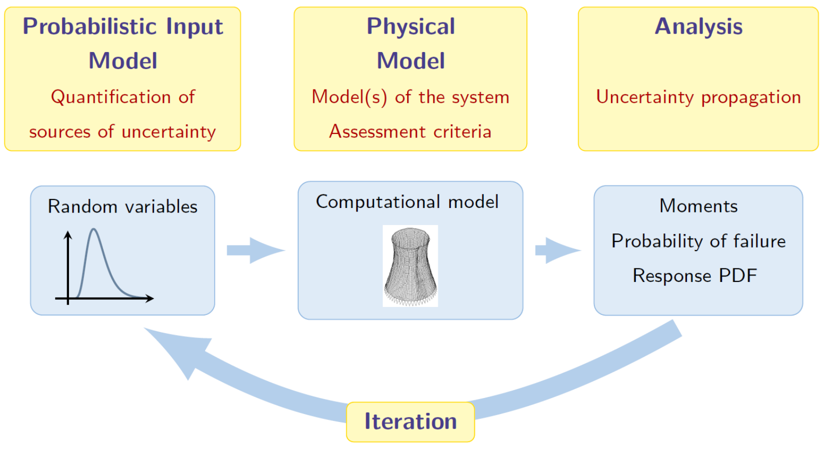


Figure 1: Visual representation of the global theoretical framework for uncertainty quantification developed by Sudret (2007); de Rocquigny et al. (2008), which gives the theoretical foundations to the UQLAB software.

The formalism used for defining such computational models in the UQLAB software framework is introduced in the UQLAB User Manual – the MODEL module . The present manual extends the framework to a more universal definition of wrappers for computational models developed in third-party software.

1.2 Formalism

The physical model can be seen as a black-box, *i.e.* a map from the space of input parameters to that of output quantities:

$$y = \mathcal{M}(x), \tag{1.1}$$

where x is the input vector of size M, y is the vector of output quantities of size O and $\mathcal{M}$ represents the computational model.
As introduced in the UQLAB User Manual – the MODEL module , UQLAB offers various means to implement such computational models. In simple applications, the computational model can be described by mathematical functions known in closed form. These can be easily coded in MATLAB through the so-called *m-functions*. However, as the complexity of the models increases, simple closed form equations may not be sufficient anymore. Numerical approximations coded in specialized software are the common practice for such cases. The scope of application is extremely broad and may encompass various science fields with different levels of abstraction, *e.g.* engineering, physics, biology, economics, etc. In engineering for instance, numerical methods such as finite elements are used to simulate the behavior of systems.
The aim of this manual is to show how third-party software running such numerical models can be linked to UQLAB for the purpose of uncertainty quantification.

1.3 Code execution from command line

In general, executing an analysis using a specialized software or code requires three steps:

- **Pre-processing**: in this step, the model to be analyzed is prepared. This requires setting all the parameters that are necessary for the solution of the underlying mathematical problem. Quite often, pre-processing is done with the help of a *graphical user interface* (GUI), which allows the user to interactively define the model. For instance, in finite element analysis, the pre-processing consists in creating a geometry, generating a mesh, specifying the analysis type and options such as the boundary conditions and the material properties and eventually specifying the quantities of interest that need to be recorded/displayed.

- **Analysis (processing)**: in this step, the constitutive equations of the underlying mathematical model are solved. This is the main part of the entire analysis. The user has very little control on this part of the execution, which can be regarded as black-box.

- **Post-processing**: in this step, the results are reviewed and interpreted by the user. This can be done through a plain text file or a graphical user interface.

The parts of the software that handle each of these steps are respectively called *pre-processor*, *solver* and *post-processor*. Even though they could be entirely independent software, in most cases they belong to the same framework.
Once the preprocessing stage is complete, the user runs the analysis (mainly using a link from the GUI) and eventually interprets the results. However, when there is a need to automate the analysis and run multiple instances, say by varying some input parameters, other approaches

need to be considered. This is usually achieved by executing an *operating system command*. This requires one or several *input file(s)* and a *command line* that executes the software in batch mode, which then generates one or several *output file(s)*.

1.3.1 Input file

The input file is the means of communication between the pre-processor and the solver. It contains all information necessary to completely define the model and the analysis. When a user pre-processes an analysis using a GUI, an input file is automatically generated. Most of the time, this is a human-readable text file written in a predefined format, *i.e.* based on *keywords*. Its structure is also intuitive, allowing for manual alteration of its content using any text-editor.

1.3.2 Command line

Most software can be easily executed from the operating system command line. In general the execution command line is specific to each software and consists of a set of simple arguments. The following two arguments are often mandatory:

- the executable whose extension often reads `.exe` ;
- the input file name whose extension depends on the associated software.

Additional arguments may be required, such as the name of the output file, verbosity, display options, etc.
In the simplest case, the command line can be in the following form:

```
<exe> <inputfilename> <argument1> <argument2> ...
```

This assumes a predefined ordering in the arguments, *e.g.* the input file is expected directly after the `<exe>` command in the above example. An example of such command line format can be found in the ABAQUS example of this manual (See Section 3.1 for more details).
Another common possibility is a the use of a more elaborate command line where the arguments are prefixed, *e.g.*:

```
<exe> -i <inputfilename> -o <outputfilename> -a <argument> -b ...
```

In this example, it is assumed that the prefix `-i` indicates the input file, `-o` indicates the output file, `-a` indicates the value of a given option and `-b` is a binary option (*e.g.* display results or not).

1.3.3 Output file

Once the analysis is run, a set of new files is usually generated by the solver. Among them is the output file where the results of the analysis are stored. Depending on the field practice, this file can be binary and in a proprietary format, *i.e.* only readable through a specific post-processor. This usually allows for easy interpretation of the results by visualization.

Alternatively, the outputs may be displayed in a human-readable text file. This is the option considered when automating computations. There are two possible cases for this output file. It may be a generic pre-formatted keyword-based file, thus allowing the user to easily spot and retrieve the quantities of interest. In some cases, the user defines exactly the structure of the output file in pre-processing by submitting a script that will write the results in the required format. In both cases, the results can be automatically retrieved through a user-defined *parser*.

1.4 Generic structure of a wrapper

The core of the UQLink module is a *code wrapper*, also known as a plug-in. A wrapper is a code that allows the execution of one program within the scope of another. To some degree, it can be seen as a "translator" from the first program to the second. In UQLink, the main program is UQLab, or more specifically Matlab, while the second is any third-party software. To create a wrapper using UQLink, three main points are needed:

- **The input format of the software that is to be connected to UQLab:** As explained above, most third-party software collect their input parameters in human-readable configuration files and are executed through a command line.
- **The executable command of the software that is to be connected to UQLab:** This is a command that is used to launch the execution of the software. In most cases, it requires as an argument the input file(s) name(s). In UQLink, this command line is executed directly from Matlab using the `system` command.
- **The output format of the software that is to be connected to UQLab:** To allow for automatic parsing of the results within UQLab, the output file should have a consistent format. A parser, herein a Matlab m-function, needs to be created to retrieve the results from the output file.

Given this information, the wrapper function of the UQLink module mainly performs the following operations:

1. Receive and interpret a set of input parameters (in practice, it is a realization of an input random vector in the context of uncertainty quantification or a point of an experimental design (DOE) in the context of surrogate modelling);
2. Create a set of input configuration files and command lines according to the third party software format.This is done in UQLink by defining a *template* in which some *markers* are set (See details in Section 2.3);
3. Execute the third-party software with the configuration files just generated;
4. Retrieve and interpret the results from the output of the executed program;

5. Reformat the results and return them in the format required by UQLab.

By carrying out these steps, the UQLab's UQLink module offers a convenient way to build a wrapper (in the format of a UQLab Model object) which can then be used in an uncertainty quantification analysis.

Chapter 2

Usage

2.1 Reference problem

This chapter will demonstrate how UQLAB can be linked to a third-party software. The reference problem consists of a simply supported beam with rectangular cross section (width b and height h) which is uniformly loaded, as illustrated in Figure 2. The quantity of interest for the analysis is the midspan deflection (vertical displacement) of the beam (denoted by V in Figure 2).

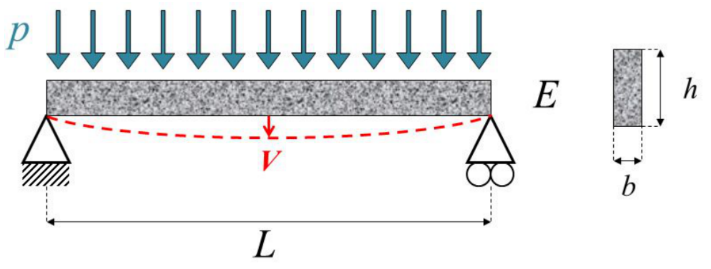


Figure 2: Simply supported beam

The underlying mathematical equation that gives the displacement for a configuration of the model reads:

$$y = \mathcal{M}(\boldsymbol{x}) = \frac{5pL^4}{32Ebh^3}, \tag{2.1}$$

where $\boldsymbol{x} = \{b,\, h,\, L,\, E,\, p\}^T$ is the vector of input parameters corresponding respectively to the beam width, height and length, the beam constitutive material Young's modulus and the uniform load. The output of interest y corresponds to the beam midspan deflection V.

2.2 Structure of the third-party software

Let us now assume that the constitutive equation defined in Eq. (2.1) is implemented in a third-party software. As explained in Section 1.4, three basic information are needed to define the wrapper:

- The input file: For this model, the input file consists of a simple text file which is structured as follow:

  ```
  % Input file for the simply supported beam model
  0.15 % b in m
  0.3 % h in m
  5 % L in m
  30E9 % E in Pa
  10000 % p in N/m
  ```

  The file starts with a header line (commented with `%`) that describes briefly its content. The five input parameters are then written separately in different lines. Each line consists of the value of the parameter followed by a blank space and a comment that gives further details about the corresponding parameter. In this example, the beam width, height and length are respectively 0.15 m, 0.3 m and 5 m, the material's Young's modulus is 30 GPa and the uniform load is 10 kN/m. This input file is named `BeamDeflection.inp`.

- The executable command: The third-party software can be run using an executable command called `myBeam.exe`. The command line is the simplest possible and reads:

  ```
  myBeam BeamDeflection.inp
  ```

  This command line assumes that, at the moment of execution, the input file is located in the current path. In general when this is not the case, the user can provide the input file together with the full path. As for the executable, it is either assumed that `myBeam.exe` is in the current path or is an *operating system environmental variable*. When none of these is true, it is also possible to use in the command line the full path to the executable. Other options related to indicating paths to different files are explained in Section 2.7.2.

- The output file: For this model, it is a simple text file containing one single value corresponding to the requested midspan displacement:

  ```
  8.0E-3
  ```

  It is assumed that this file is called `BeamDeflection.out`.

The different steps needed to build the UQLINK object are defined in the following sections.

2.3 Preparing the template

The input template is the generic file from which actual input files that will be used to run the code for each realization of the parameters are derived. In this sense, it is pre-formatted by setting up *markers* in lieu of parameters that need to be modified during the UQLAB anlysis. The markers in UQLINK are made of three elements, namely two *delimiters* and a *variable name*. The delimiters are strings that start and end the marker (see options `.Marker` in

Table 2). The variable name is a string that will be concatenated with incremental 4-digit-numeric counters.

Note: By default, the two delimiters are `'<'` and `'>'` and the variable name is `'X'`.

As a whole, in this example, the name of the parameters in the input template will be `<X0001>`, `<X0002>`, etc...
The first step in creating the template is to make a copy of the original input file and renaming it by adding another extension to the current one, *e.g.* `.tpl`. This allows one to prevent UQLAB from making any action or modification that may corrupt the original input file.
Once this template is created, the user has to set the markers. As this is problem-dependent, this has to be done by editing manually the template file. For the simply supported beam example, the tagged parameters are the five parameters defined above. The template, named `BeamDeflection.inp.tpl`, is therefore marked as follows:

```
% Input file for the simply supported beam model
<X0001> % b in m
<X0002> % h in m
<X0003> % L in m
<X0004> % E in Pa
<X0005> % p in N/m
```

2.4 Retrieving the results

A MATLAB function that reads the output file should be provided. This function should take as input a string of characters or a cell array of strings which corresponds to the name of the output files. The latter is for the case when the outputs are retrieved from many files (See Table 3). The outputs of the function should be scalars or row vectors. For the simply supported beam example, the following code can be used:

```
function Y = uq_readOutput(outputfile)
% Read the single line of the file , which corresponds to the
% sought midspan beam deflection
Y = dlmread(outputfile) ;
end
```

2.5 Creating the UQLINK object

A UQLINK MODEL object is created in UQLAB with the command `uq_createModel()`. The basic and mandatory model options are:

- **.Command**: The execution command which may or may not include a full path;
- **.Template**: The name of the template file ;

- **.Output**: A structure containing the parser (`.Output.Parser`) and the name(s) of the output file(s) (`.Output.FileName`). The parser is the MATLAB function that reads the output of the third party software, retrieves the results and formats them into a row vector (possibly with one entry) for further processing in UQLAB. The user should make sure that this function belongs to the current MATLAB path.

For the simply supported beam example, the following code may be used:

```
% Start the framework (if not already started)
uqlab ;
% Define the model options
modelopts.Type = 'UQLink' ;
modelopts.Command = 'myBeam BeamDeflection.inp' ;
modelopts.Template = 'BeamDeflection.inp.tpl' ;
modelopts.Output.Parser = 'uq_readOutput' ;
modelopts.Output.FileName = 'BeamDeflection.out' ;

% Create and add the model to UQLab
myModel = uq_createModel(modelopts) ;
```

Note: The strings given in `modelopts` are all case-sensitive.

In practice, for this example UQLINK will first create a configuration file with values corresponding to different input parameters that need to be evaluated. There will be one file for each set of input parameters. Their names are formed by appending a numeric counter to the original input file name. In this example, the generated input files will be named `BeamDeflection000001.inp`, `BeamDeflection000002.inp`, etc. Then for each input file, a command line will be generated and executed. For the first input file, the command that will be submitted to the system by UQLINK is:

```
myBeam BeamDeflection000001.inp
```

Finally, the corresponding `outputfilename` variable defined in the parser function will be given the value `BeamDeflection000001.out`. In general UQLINK generates the input file name by appending a numeric counter similar to the input.

Note: UQLINK always assumes that the actual output files generated during execution have the same basis name as given in `.Output.FileName` to which is appended a numeric counter (similar to the corresponding input file). During execution, if the output file name is found either in the template file or in the command line, it is replaced so as to correpsond to the output file name exepcted by UQLINK, *i.e.* with a numeric counter.

2.6 Evaluating the model

The `uq_evalModel` command can be used to directly evaluate the model response on a specified matrix of inputs X:

```
Y = uq_evalModel(X) ;
```

The above command assumes that the UQLINK MODEL object is the active one in the current UQLAB session. To explicitely specify which model to run, the following command should be used:

```
Y = uq_evalModel(myModel, X) ;
```

Assuming that the input is of size $N \times M$, the output Y may be formatted in one of the following ways:

- Vector of size $N \times 1$, if the post-processing of the third-party code delivers a single scalar value;
- Matrix of size $N \times O$, if the post-processing of the third-party code delivers a row vector of length O.

When multiple outputs of different types are used, the following command should be considered:

```
[Y1,Y2,...] = uq_evalModel(myModel, X) ;
```

The user should make sure that this is reflected in the `uq_readOutput` file.

2.7 Advanced options

2.7.1 Name of the model and results saving

A name can be given to a UQLINK MODEL as for any UQLAB MODEL. This can be achieved using the option `.Name`, *e.g.*

```
modelopts.Name = 'Beam Deflection Wrapper'
```

By default, the name of the model is `Model k` where `k` is an integer indicating that the model is the k-th public MODEL created in the current UQLAB session.
This information is used when evaluating the model. In fact, UQLINK always save the results of the different runs as they are processed. After each computation, the concatenated results are saved in a matrix located in the current execution path and named after the MODEL (*e.g.*, for the simply supported beam, this variable is named `BeamDeflectionWrapper.mat`). This matrix contains the inputs and outputs that have been processed so far. They are named respectively `uq_ProcessedX` and `uq_ProcessedY` when there is only one output. In case of multiple outputs, the matrices are named `uq_ProcessedY1`, `uq_ProcessedY2`, `uq_ProcessedY3`, etc. Such a procedure allows one to retrieve available results in the case a crash occurs during execution.

Note: When creating the name of the `.mat` file using the MODEL name, any existing blank space is deleted.

2.7.2 Paths specifications

By default, it is assumed that the executable and input template are directly accessible with the options provided above. For the executable, this may be because the operating system can automatically find the directory where it is located (the executable is an operating system environmental variable) or simply because the full path has been specified by the user. For the template, it is assumed that the file belongs to the current MATLAB path at the time it is called or that the user has specified a full path in the command line. When this is not the case, one may directly specify the following options:

- `.ExecutablePath`: This is the path to the directory where the executable command is located;
- `.ExecutionPath`: This is the path to the directory where the input template, and any other file needed by the executable command to properly run, are located. This will also be the directory where the execution is actually processed.

As an example, let us assume that, on a windows platform, the executable is located in `C:\users\username\software\bin` and the execution path is `C:\user\username\Application\Beam` . The user can include this information by using the following commad:

```
modelopts.ExecutablePath = 'C:\users\username\software\bin' ;
modelopts.ExecutionPath = 'C:\users\username\Application\Beam' ;
```

The command line for the first example above will therefore read:

```
C:\users\username\software\bin\myBeam ...
 C:\users\username\Application\Beam\BeamDeflection000001.inp
```

Likewise, the `outputfilename` of the parser will be `C:\users\username\Application\Beam\BeamDeflection000001.out`.

Note: By default, these two options `.ExecutablePath` and `.ExecutionPath` are assumed to be empty characters, *i.e.* `''`.

2.7.3 Files counter

The user can specify the number of digits used for numbering the input and output files. By default 6 digits are considered. To produce files with a different number of digits, say 3, the following code can be used:

```
modelopts.Counter.Digits = 3
```

The resulting files, say for the first input parameter, will be `BeamDeflection001.inp` and `BeamDeflection001.out`.

It is also possible to specify the starting point of the counter. By default, the numbering starts with 1 and each new file number is incremented by 1. To offset the first input file number, the following code can be used:

```
modelopts.Counter.Offset = 1001
```

In this case the corresponding input files will be `BeamDeflection001001.inp`, `BeamDeflection001002.inp`, etc. The output files are assumed to be numbered in the same system.

Note: The user should make sure that the `.Digit` and `.Offset` options are consistent. For instance, setting `.Counter.Digits = 3` and `.Counter.Offset = 1001` would result in an error as the integer 1001 cannot be written in 3 digits.

2.7.4 Format specification of the variables in the input file

The input realizations are written in the input file as strings according to a given format. This format may depend on the software specifications. By default, all numeric values are written with an exponential notation using one and six digits respectively before and after the decimal point. This corresponds to the MATLAB format specification `'%1.6E'`. Another format may be specified by using the `.Format` option, *e.g.*:

```
Mopts.Format = '%f' % Fixed-point notation
```

This command will affect the same format to all the variables. When one needs to specify different formats to each variable, a cell structure can be used. For instance, if the input is two-dimensional, *i.e.* $M = 2$, a formatting option can be:

```
Mopts.Format = {'%1.8e','%2.6f'}
```

For more details on the formatting operators, the reader may refer to the MATLAB documentation.

2.7.5 Archiving new files

During the execution of the third-party software, a number of files are generated, among which the output file where the quantities of interest are written and auxiliary files such as log files. When the number of input realizations is large, this may lead to a wide number of generated files that may either make spotting of useful files difficult or may uselessly waste storage space on the disk. UQLINK provides an automatic way of handling the files generated during processing. The provided options can be set in the field `.Archiving` (See Table 5). The following options for the archiving action are provided:

- Archiving the files: All the files are kept but moved to a specific location defined by the user. Assuming that this repository is `SaveRepository`, the following folders are created:

  - `SaveRepository\UQLinkInput`, where all the generated input files are saved, *e.g.* `BeamDeflection000001.inp`, `BeamDeflection000002.inp`, etc.
  - `SaveRepository\UQLinkOutput`, where all the output files are saved, *e.g.* `BeamDeflection000001.out`, `BeamDeflection000002.out`, etc.
  - `SaveRepository\UQLinkAux`, where all the auxiliary files are saved, *e.g.* `BeamDeflection000001.log`, `BeamDeflection000001.msg`, `BeamDeflection000002.log`, `BeamDeflection000002.msg`, etc.

  The corresponding option argument is:

  ```
  Mopts.Archiving.Action = 'save'
  ```

- Deleting all the files: This means that at the end of the analysis the only remaining file will be the `.mat` file gathering the processed results. The corresponding option argument is

  ```
  Mopts.Archiving.Action = 'delete'
  ```

- Doing nothing: All the files are kept and stay where they have been generated. The corresponding option argument is:

  ```
  Mopts.Archiving.Action = 'none'
  ```

Note: By default, UQLINK considers the saving action, *i.e.* `.Archiving.Action = 'save'`.

Furthermore, when saving is enabled other options can be specified by the user:

  - The name of the repository where the data will be saved: This option can be set by the user using the command `.Archiving.FolderName`. For instance on a windows platform, to save the results in the folder `C:\user\username\Application\BeamResults`, the following code can be used:

    ```
    modelopts.Archiving.FolderName = ...
    'C:\users\username\Application\BeamResults'
    ```

    By default, the save repository is created under the execution path and is named after the MODEL (the name of the folder corresponds to the MODEL name with all blank spaces deleted).
  - Compression of the results: This option is a logical which decides whether the archiving repository should be compressed or not. By default, compression is enabled. In this case, the `.zip` folder is created in the archiving repository. The uncompressed data are then erased. To disable compression, the following option can be used:

```
modelopts.Archiving.Zip = false ;
```

2.7.6 Multiple input files

In some cases, the model parameters may be located in different input files that need to be accessed by the third-party software. In such cases, the options described above need to be slightly adapted to account for the multiple input files. Let us for instance assume the following command line, which requires two input files:

```
myExecutable.exe file1.inp file2.dat
```

To properly build the wrapper, the user should prepare templates for each file. For the example above, the `.Template` option would read:

```
Mopts.Template = {'file1.inp.tpl', 'file2.dat.tpl'}
```

Note: Even if an executable requires multiple input files, a template should only be created for those which contains parameters that need to be modified during the analysis. The other files (which remain unchanged for all runs) are considered only as additional arguments of the command line.

2.7.7 Multiple output files

In some cases, the quantities of interest can be saved in different output files. To define these output files in UQLINK, the `.Output.FileName` should be a cell array where each element is one output file. For instance, if a code returns the following two output files `output1.out` and `output2.out`, the following code can be used to specify them:

```
modelopts.Output.FileName = {'output1.out', 'output2.out'}
```

Note: In this case, the parser function should have as input argument a cell array where the output files are defined as in the command `.Output.FileName`.

2.7.8 Recovering failed results

While evaluating a UQLINK MODEL, various errors may occur during the execution of the third-party software or when recovering the results. In such cases, UQLINK catches the error, issues a warning and proceeds to the next simulation. The responses corresponding to failed simulations are set to `NaN`, both in the output vector(s) (of `uq_evalModel`) and in the variable `uq_ProcessedY` (or `uq_ProcessedY1`, `uq_ProcessedY2`, etc. in case of multiple outputs) of the MODEL's saved `.mat` file.
Failed simulations can be run again by UQLINK using the `'recover'` argument. When a UQLINK MODEL has already been evaluated, the following command:

```
Y = uq_evalModel(X,'recover') ;
```

recovers the MODEL's saved `.mat` file, looks up for all the lines with `NaNs` in the output and runs again the corresponding simulations. The returned vector `Y` is the full vector of outputs (previously simulated and new runs).

Note: The given vector `X` should be exactly the same used in the previous MODEL evaluation. When this is not the case, an error is returned.

When necessary, *e.g.* if the `.mat` file has been moved or renamed, the user can specify the file in which UQLINK should search for the previous results:

```
Y = uq_evalModel(X, 'recover', 'RecoverySource') ;
```

In this case, `'RecoverySource'` should be an accessible `.mat` file whose `uq_ProcessedX` is equal to the given `X`.
Finally, the user can directly specifies a list of simulation numbers that should be run again:

```
Y = uq_evalModel(X,'recover', RunList) ;
```

`RunList` is a vector with digits not larger than the length of the vector `X`.

2.7.9 Resuming an analysis

In some cases, a UQLINK MODEL evaluation can be stopped before completion. When the already run simulations are still valid, the user can resume the evaluation where it stopped using the argument `'resume'`:

```
Y = uq_evalModel(X,'resume') ;
```

In this case, UQLINK first compares `uq_ProcessedX` and the given `X`. If all the lines of `uq_ProcessedX` are equal to an upper part of `X`, the evaluation is resumed, starting from the first not-yet-evaluated line of `X`.
When necessary, *e.g.* the `.mat` file has been moved or renamed, the user can specify the file in which UQLINK should search for the results:

```
Y = uq_evalModel(X, 'recover', 'RecoverySource') ;
```

In this case, `'RecoverySource'` should be an accessible `.mat` file whose `uq_ProcessedX` is equal to an upper part of the given `X`.

Chapter 3

Examples

3.1 Truss analysis with ABAQUS

3.1.1 ABAQUS FEA software

ABAQUS FEA is a general-purpose commercial finite element analysis software suite developed and distributed by DS Simulia (Dassault Systèmes, 2017) . It consists of different core products, among which:

- ABAQUS/CAE: a graphical user interface (GUI) software used for pre-processing (*i.e.* creating the model geometry and assigning the analysis options) and for post-processing (*i.e.* visualizing the results of the finite element analysis).
- ABAQUS/STANDARD: a solver where the actual finite element analysis is carried out.

Other software of the suites can be used for special applications, *e.g.* computational fluid dynamics or computational electromagnetics. For the example in this manual, only ABAQUS/STANDARD is used (Version 6.14).

3.1.2 Presentation of the model

In this chapter we study a ten-bar, linear-elastic truss structure as introduced by Wei and Rahman (2007). The structure is illustrated in Figure 3. Each member has a Young's modulus of 10^7 psi. The structure is subjected to concentrated loads applied at nodes 2 and 4, namely $F_2 = F_4 = 100,000$ lb. The cross-sectional areas of the 10 bars are modelled independently by truncated Gaussian random variables $X_i \sim \mathcal{N}_{[10^{-5},+\infty]}\left(2.5,\, 0.5^2\right), i = \{1, \ldots, 10\}$ (expressed in in^2). The quantity of interest in the outputs is the maximal displacement u which occurs at node 2.

3.1.3 UQLINK input file

To create a UQLINK MODEL object for the above truss, the following code can be used:

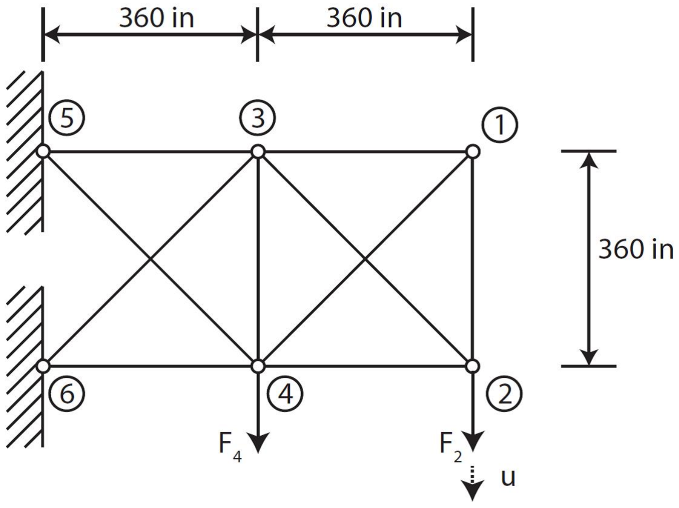


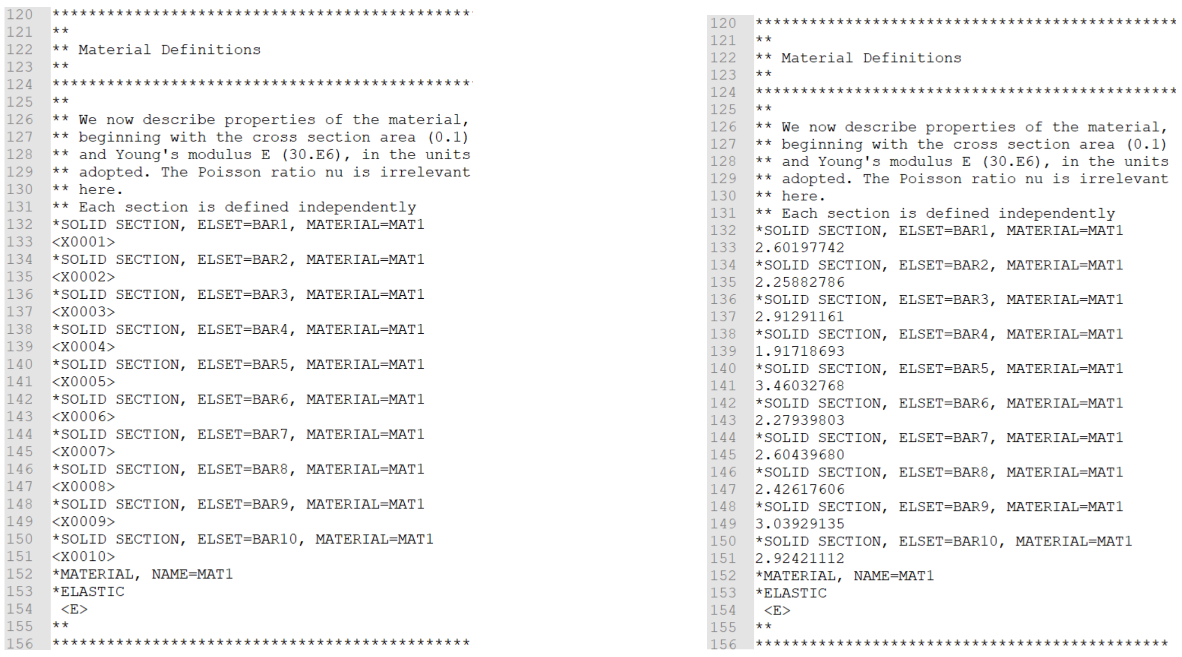
Figure 3: Ten-bar truss structure.

```
% Start the framework (if not already started)
uqlab
% Model type
Mopts.Type = 'UQLink' ;
% Mandatory options
Mopts.Command = ...
'C:\SIMULIA\Abaqus\Commands\abaqus -job TenBarTruss interactive' ;
Mopts.Template = 'TenBarTruss.inp.tpl' ;
Mopts.Output.FileName = 'TenBarTruss.dat' ;
Mopts.Output.Parser = 'uq_readTenBarTrussOutput' ;
% Non-mandatory options
Mopts.ExecutionPath = fullfile(uq_rootPath, 'Examples', ...
 'UQLink', 'Abaqus_Truss') ;
Mopts.Format = {'%1.8f'} ;
% Create the model
AbaqusModel = uq_createModel(Mopts) ;
```

The `.Command` field shows the execution command that is needed to run the ABAQUS job. The execution path, that is the folder in which the analysis will be run, is given by the option `.ExecutionPath`. Here the MATLAB command `fullfile` is used to set the path by concatenating various elements starting from the UQLAB root path given by the UQLAB command `uq_rootPath`. In the current case (windows platform), it would result in the string `'C:\users\username\UQLabCore\Examples\UQLink\Abaqus_Truss'`.

The cross-sectional areas of the truss bars are written in the input file using a floating-point number format with 8 digits as indicated with the option `.Format`. Figure 4 shows on the left side the markers of the template file, namely `<X0001>`, `<X0002>`, etc. and on the right one an example of input realization using the specified format.

The MATLAB function `uq_readTenbarTrussOutput.m` is provided to read the output file which is specified in UQLINK through the option `.Output.FileName`. The output requested



```
**********************************************
**
** Material Definitions
**
**********************************************
**
** We now describe properties of the material,
** beginning with the cross section area (0.1)
** and Young's modulus E (30.E6), in the units
** adopted. The Poisson ratio nu is irrelevant
** here.
** Each section is defined independently
*SOLID SECTION, ELSET=BAR1, MATERIAL=MAT1
<X0001>
*SOLID SECTION, ELSET=BAR2, MATERIAL=MAT1
<X0002>
*SOLID SECTION, ELSET=BAR3, MATERIAL=MAT1
<X0003>
*SOLID SECTION, ELSET=BAR4, MATERIAL=MAT1
<X0004>
*SOLID SECTION, ELSET=BAR5, MATERIAL=MAT1
<X0005>
*SOLID SECTION, ELSET=BAR6, MATERIAL=MAT1
<X0006>
*SOLID SECTION, ELSET=BAR7, MATERIAL=MAT1
<X0007>
*SOLID SECTION, ELSET=BAR8, MATERIAL=MAT1
<X0008>
*SOLID SECTION, ELSET=BAR9, MATERIAL=MAT1
<X0009>
*SOLID SECTION, ELSET=BAR10, MATERIAL=MAT1
<X0010>
*MATERIAL, NAME=MAT1
*ELASTIC
 <E>
**
**********************************************
```

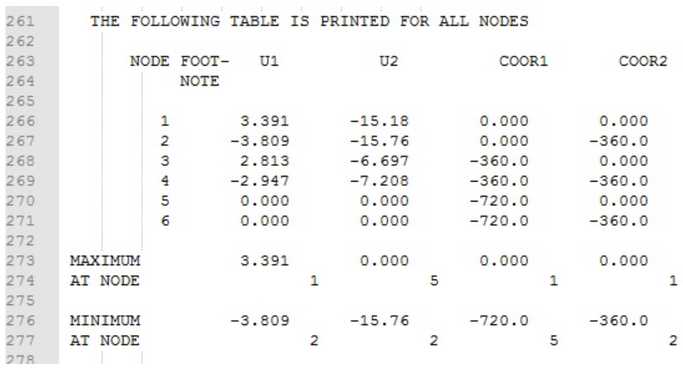
(a) Template

```
***********************************************
**
** Material Definitions
**
***********************************************
**
** We now describe properties of the material,
** beginning with the cross section area (0.1)
** and Young's modulus E (30.E6), in the units
** adopted. The Poisson ratio nu is irrelevant
** here.
** Each section is defined independently
*SOLID SECTION, ELSET=BAR1, MATERIAL=MAT1
2.60197742
*SOLID SECTION, ELSET=BAR2, MATERIAL=MAT1
2.25882786
*SOLID SECTION, ELSET=BAR3, MATERIAL=MAT1
2.91291161
*SOLID SECTION, ELSET=BAR4, MATERIAL=MAT1
1.91718693
*SOLID SECTION, ELSET=BAR5, MATERIAL=MAT1
3.46032768
*SOLID SECTION, ELSET=BAR6, MATERIAL=MAT1
2.27939803
*SOLID SECTION, ELSET=BAR7, MATERIAL=MAT1
2.60439680
*SOLID SECTION, ELSET=BAR8, MATERIAL=MAT1
2.42617606
*SOLID SECTION, ELSET=BAR9, MATERIAL=MAT1
3.03929135
*SOLID SECTION, ELSET=BAR10, MATERIAL=MAT1
2.92421112
*MATERIAL, NAME=MAT1
*ELASTIC
 <E>
**
**********************************************
```

(b) Actual input file

Figure 4: Input template and actual file showing the markers and the input realizations.

by the ABAQUS input file is written using the own ABAQUS default formatting as shown in Figure 5. The `readTenbarTrussOutput.m` function is simply processing the output text file so as to retrieve the requested output, *i.e.* the variable `U2` of node `2`. The default ABAQUS formatting for displacements, is a 4-digit floating-point number.



```
   THE FOLLOWING TABLE IS PRINTED FOR ALL NODES

        NODE FOOT-   U1           U2           COOR1        COOR2
             NOTE

          1        3.391       -15.18        0.000        0.000
          2       -3.809       -15.76        0.000       -360.0
          3        2.813       -6.697       -360.0        0.000
          4       -2.947       -7.208       -360.0       -360.0
          5        0.000        0.000       -720.0        0.000
          6        0.000        0.000       -720.0       -360.0

 MAXIMUM           3.391        0.000        0.000        0.000
 AT NODE                  1            5            1            1

 MINIMUM          -3.809       -15.76       -720.0       -360.0
 AT NODE                  2            2            5            2
```

Figure 5: Extract of the ABAQUS output file for the ten bar truss problem showing the nodal displacements.

Note: For advanced examples, ABAQUS offers the option of using `python` routines to get the outputs in text files which only contain the values of interest in a specific suitable format without any other information. This is usually a good practice when the default output file is (memory-wise) large.

3.1.4 Evaluating the model

The evaluation of the model is similar to any UQLAB model, see for instance UQLAB User Manual – the MODEL module , UQLAB User Manual – Polynomial chaos expansions or UQLAB User Manual – Kriging (Gaussian process modelling) . The following lines of code are used to evaluate the model on 200 realizations of the input variables:

```
%Input marginals
for ii = 1:10
Iopts.Marginals(ii).Name = ['A',num2str(ii)]; %cross section areas
Iopts.Marginals(ii).Type = 'Gaussian';
Iopts.Marginals(ii).Moments = [2.5, 0.5]; % in^2
Iopts.Marginals(ii).Bounds = [1e-5, inf];
end
% Create input object
myInput = uq_createInput(Iopts);
% Experimental design of size 200
Xtruss = uq_getSample(200,'LHS') ;
% Evaluate the model
Ytruss = uq_evalModel(AbaqusModel, Xtruss) ;
```

An INPUT object with ten independent variables following truncated Gaussian distributions is first built. Then a set of 200 points are generated using a Latin hypercube sampling scheme with the command `uq_getSample`. The input matrix $\boldsymbol{X}$ is then of size 200×10. The model is finally evaluated using the command `Ytruss = uq_evalModel(AbqusModel, Xtruss)`. The resulting vector $\boldsymbol{Y}$ is of size 200×1. Note that if `AbaqusModel` is the active UQLAB MODEL, *e.g.* it is the last created MODEL in the workspace, the first argument is not necessary, *i.e.* `Ytruss= uq_evalModel(Xtruss)` would be provide the same results.

3.1.5 Further use of the UQLINK model

Once this model is created, it can be used for any analysis as any UQLAB MODEL object. To show some application examples, let us consider the reliability analysis of this structure as carried out in Wei and Rahman (2007); Bae and Alyanak (2016). For this purpose, a limit-state function $g(\boldsymbol{X}) = 18 - u(\boldsymbol{X})$ is defined. The aim of the analysis is to estimate the structure's failure probability related to the vertical displacement of node 2 being greater than 18 inches. We consider here a Monte Carlo simulation based on a polynomail chaos expansion (PCE) approximation of the limit-state function, as provided in the UQLAB User Manual – Structural reliability (Rare event estimation) .

3.1.5.1 Experimental design

To build the PCE model, an experimental design of size 200 is first generated. This can be achieved here by using the codes in Section 3.1.4. For more insight into this model, kernel density smoothing is used to plot an empirical probability density function (PDF) of the output given the data available in the experimental design.

```
% Create the kernel smoothing density function
[f,xi] = ksdensity(Ytruss) ;
% Plot the resulting KS density
plot(xi,f) ;
```

Figure 6 shows the resulting kernel density plot.

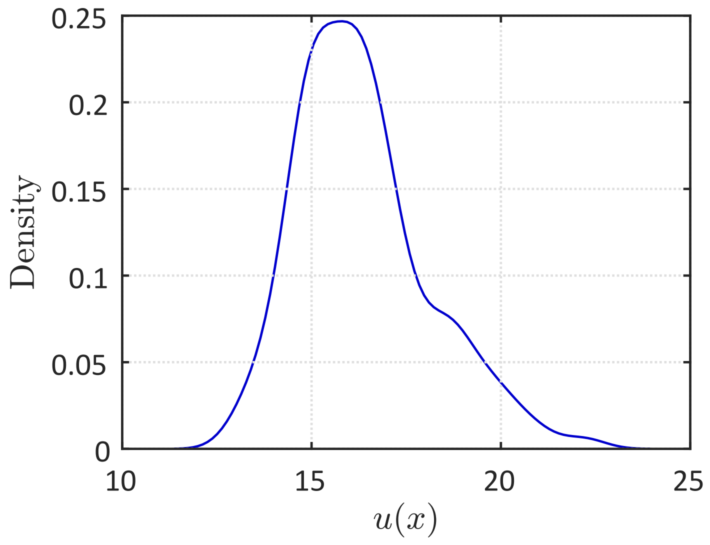


Figure 6: PDF of the quantity of interest obtained using kernel smoothing on the experimental design.

3.1.5.2 PCE model and sensitivity analysis

A PCE model can then be built given the generated data points. The following code can be used to this end:

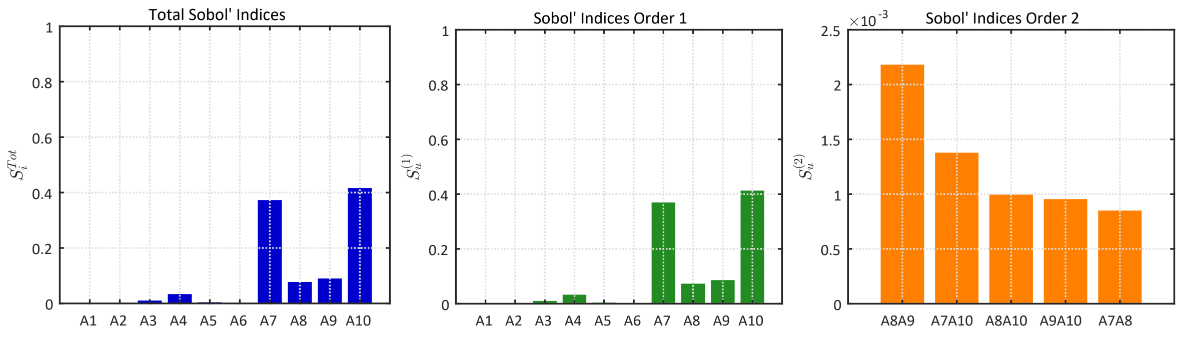
```
% Model type
metaopts.Type = 'metamodel' ;
metaopts.MetaType = 'PCE' ;
% PCE options
metaopts.Degree = 2:10 ;
metaopts.ExpDesign.X = Xtruss ;
metaopts.ExpDesign.Y = Ytruss ;
% Create the model
myPCE = uq_createModel(metaopts) ;
```

The user can refer to UQLAB User Manual – Polynomial chaos expansions for more details on how to build a PCE model in UQLAB.

A global sensitivity analysis can be run to assess which parameters is influencing the more the output variability. Sobol' indices are a popular global sensitivity analysis technique. They represent the relative portion of the input variance and their combination into the output variance. They are usually estimated by simulation, which may be expensive. An interesting property of PCE is that they allow for an analytical expression of Sobol' indices. In UQLAB, PCE-based Sobol' indices can be computed using the following code:

```
PCESobol.Type = 'Sensitivity';
PCESobol.Method = 'Sobol';
PCESobol.Sobol.Order = 2;
PCESobolAnalysis = uq_createAnalysis(PCESobol) ;
```

When a PCE model is defined(last MODEL defined, which is the active one in the workspace), UQLAB uses it to compute analytically the Sobol' indices. The results of the analysis for this example are shown in Figure 7. It can be observed that roughly four variables explain almost all the output variability. Furthermore, the sectional areas of the bars 7 and 10 explain almost 80% of the output variability. The second order indices are negligible.



(a) Total order (b) First order (c) Total order

Figure 7: Sobol' indices for the ten bar truss example.

3.1.5.3 Monte Carlo simulation using PCE approximation

Once this model is built, a crude Monte carlo simulation is performed to estimate the structure's failure probability:

```
% Model type
MCSopt.Type = 'Reliability' ;
MCSopt.Method = 'MCS' ;
% LKimit-state surface definition
MCSopt.LimitState.Threshold = 18;
MCSopt.LimitState.CompOp = '>=' ;
% Run the analysis
myMCS = uq_createAnalysis(MCSopt) ;
```

Note that by default the relibaility analysis uses the active MODEL, in this case the PCE model, to evaluate the failure probability. For more details on how to run a reliability analysis in UQLAB, the user may refer to UQLAB User Manual – Structural reliability (Rare event estimation) .

The convergence curve of the Monte carlo simulations is shown in Figure 8. The resulting failure probability is estimated to be 0.1384 which is relatively close to the reference solution (0.1394) given in Wei and Rahman (2007); Bae and Alyanak (2016). This reference solution is computed by the authors using a crude Monte Carlo simulation on the original truss model with $1,000,000$ simulations.

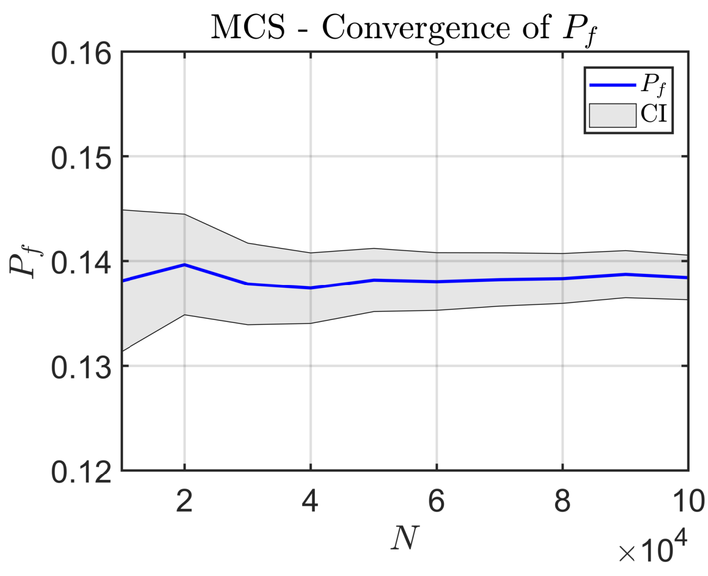


Figure 8: Convergence of the Monte Carlo simulation for the reliability analysis of the ten bar truss problem.

Note: Polynomial chaos expansions and Monte Carlo Simulation are used for estimating the failure probability because the final result ($P_f \approx 0.134$) is large. This approach is not recommended for probabilities smaller than 10^{-3}. In that case, other reliability methods such as AK-MCS shall be preferred

3.2 Pushover analysis using OpenSees

3.2.1 OpenSees Software

OPENSEES is an open-source software framework originally developed by the Pacific earthquake engineering research center (PEER). It allows users to create finite element models for simulating the response of structural and geotechnical systems subjected to earthquakes. The framework is specialized in performance-based design and non-linear analysis. More information can be found in McKenna et al. (2010) or on the software website (last accessed on June 13th, 2018).

3.2.2 Presentation of the model

This example presents a pushover analysis of a two-story, one-bay structure using OPENSEES. The pushover analysis is a static nonlinear structural analysis method widely used in performance-based design. It investigates a structure's lateral deformation under gravity loads and increasing lateral loads distributed according to a predefined pattern. The result of the analysis is a displacement *vs.* force curve (*pushover-curve*). This curve is non-linear due to the formation of local plastic hinges in the structure. Once this curve has been created, it can be used to determine the structural performance during earthquakes in conjunction with acceleration displacement response spectra (ADRS).
The structure considered here is made of beam elements connected by zero-length rotational spring elements that capture the structure's non-linearity (plastic hinges) as illustrated in the schematic representation of Figure 9. Details on the modeling and assumptions can be found in OPENSEES user's manual on the following page (last accessed on June 13th, 2018). Lateral loads are applied to the frame. The pushover analysis is performed using a displacement-controlled scheme with a lateral force distribution $f(h)$ and a maximum displacement $u_{\max}$ equal to 10% of the structure's height H.

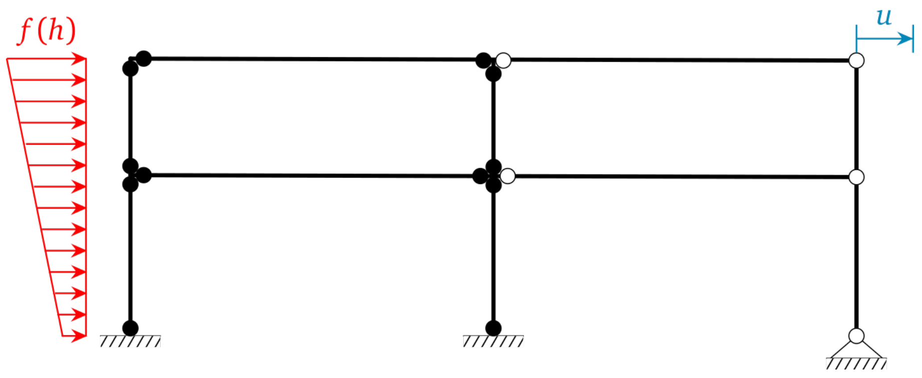


Figure 9: Schematic representation of the frame structure for the pushover analysis. The empty circles (○) represent ideal hinges, while the filled circles (●) stand for the non-linear hinges (for details see OPENSEES online user's page).

The nonlinear behaviour of the springs is modelled by a bilinear hysteretic law where the springs behave linearly until a specified yield moment. These yield moments of the individual structural components play an important role on the force-displacement behaviour of the whole structure. In this probabilistic analysis, they are modelled by the probability distributions specified in Table 1.

Table 1: Probabilistic model for the pushover analysis.

Parameter	Distribution	Mean	COV
Columns yield moment - $M_{y_{\text{col}}}$ (in ksi)	Lognormal	$20,350$	0.1
Beams yield moment - $M_{y_{\text{beam}}}$ (in ksi)	Lognormal	$10,938$	0.1

3.2.3 UQLINK input file

The UQLINK model can be created using the following code:

```
% Start the framework (if not already started)
uqlab ;
% Model type
Mopts.Type = 'UQLink' ;
% Mandatory options
Mopts.Command = 'OpenSees pushover_concentrated.tcl' ;
Mopts.Template = 'pushover_concentrated.tcl.tpl' ;
Mopts.Output.Parser = 'uq_readOutput_OpenSees_Pushover' ;
Mopts.Output.FileName = 'Vbase.out' ;
% Non-mandatory options
Mopts.ExecutionPath = fullfile('uq_rootPath', 'Examples', ...
'UQLink','OpenSees_Pushover') ;
Mopts.Format = {'%1.8f'} ;
% Create the model
OpenSeesModel = uq_createModel(Mopts) ;
```

The `.Command` field shows the execution command that is needed to run OPENSEES. The execution path, that is the folder in which the analysis will be run, is given by the option `.ExecutionPath`. Here the MATLAB command `fullfile` is used to set the path by concatenating various elements starting from the UQLAB root path given by the UQLAB command `uq_rootPath`. In this case (windows platform), it results in the string
`'C:\users\username\UQLabCore\Examples\UQLink\OpenSees_Pushover'`.
Finally, the quantity of interest, *i.e.* the base shear, can be retrieved using an output file named `Vbase.out`.

3.2.4 Retrieving the results

The `uq_readOutput_OpenSees_Pushover` function shown below is used to read the results of the analysis. In this example, the output files consist of simple text files with the time histories of the different forces components for each column. Using the MATLAB built-in function `dlmread` allows one to retrieve these data which are then saved in 3240×6 matrices. The final quantities of interest are the three first rows of the read matrix, which each corresponds to a vector of size 3240×1.

```
function [Y1,Y2,Y3] = uq_readOutput_OpenSees_Pushover(outputfile)
% Base shear
Vbase = dlmread(outputfile) ;

Y1 = Vbase(:,1)' ;
Y2 = Vbase(:,2)' ;
Y3 = Vbase(:,3)' ;
end
```

3.2.5 Evaluating the model

The evaluation of the model is similar to any UQLAB model, see for instance UQLAB User Manual – the MODEL module , UQLAB User Manual – Polynomial chaos expansions or UQLAB User Manual – Kriging (Gaussian process modelling) . The following code is used to generate the input object corresponding to the problem defined above and then to evaluate the model on 50 realizations of the input variables:

```
%Input marginals
Iopts.Marginals(1).Name = 'Mycol_12' ; %Yield momment for columns
Iopts.Marginals(1).Type = 'Lognormal';
Iopts.Marginals(1).Moments = [20350, 2035];
Iopts.Marginals(2).Name = 'Mybeam_23'; %Yield moment at plastic hinges
Iopts.Marginals(2).Type = 'Lognormal';
Iopts.Marginals(2).Moments = [10938, 1093.8];
% Create the input object
myInput = uq_createInput(Iopts);

% Generate input data
X = uq_getSample(50) ;
% Evaluate the model
[Y1,Y2,Y3] = uq_evalModel(X) ;
% The total base shear is the sum of the base shears at all bearings:
baseShear = Y1 + Y2 + Y3;
% We normalize it by the total structural weight
%(Floor2Weight+Floor3Weight):
baseShearNorm = abs(baseShear)/(500+590);
```

An input object with two independent lognormal variables is first built. Then a set of 50 points is generated by Monte Carlo sampling using the command `uq_getSample`. The input matrix $\boldsymbol{X}$ is of size 50×2. The model is finally evaluated using the command `Y = uq_evalModel(X)`. The resulting vectors `Y1`, `Y2` and `Y3` are of size 50×3240. They correspond to the base shear at each of the structure's bearings. The sum of these base shears can then be used to compute the so-called pushover-curves illustrated in Figure 10. In earthquake engineering applications, these curves are usually post-processed to find out the critical moment when the force starts to decrease as the displacement is increased.

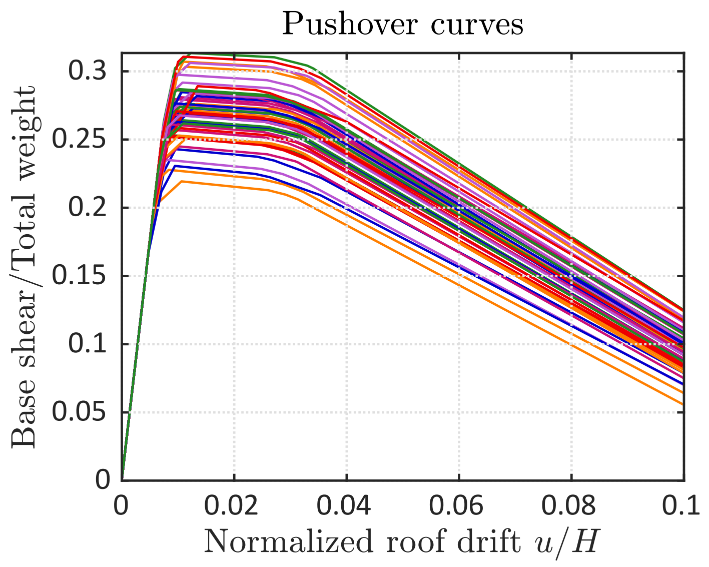


Figure 10: Pushover-curves corresponding to the 50 input realizations.

Chapter 4

Reference List

How to read the reference list

Structures play an important role throughout the UQLAB syntax. They offer a natural way to group configuration options and output quantities semantically. Due to the complexity of the algorithms implemented, it is not uncommon to employ nested structures to fine-tune inputs/outputs. Throughout this reference guide, we adopt a table-based description of the configuration structures.
The simplest case is given when a field of the structure is a simple value/array of values:

Table X: `Input`			
●	`.Name`	String	A description of the field is put here

which corresponds to the following syntax:

```
Input.Name = 'My Input';
```

The columns correspond to name, data type and a brief description of each field. At the beginning of each row a symbol is given to inform as to whether the corresponding field is mandatory, optional, mutually exclusive, etc. The comprehensive list of symbols is given in the following table:

●	Mandatory
□	Optional
⊕	Mandatory, mutually exclusive (only one of the fields can be set)
⊞	Optional, mutually exclusive (one of them can be set, if at least one of the group is set, otherwise none is necessary)

When one of the fields of a structure is a nested structure, we provide a link to a table that describes the available options, as in the case of the `Options` field in the following example:

Table X: Input			
●	.Name	String	Description
□	.Options	Table Y	Description of the Options structure

Table Y: Input.Options			
●	.Field1	String	Description of Field1
□	.Field2	Double	Description of Field2

In some cases an option value gives the possibility to define further options related to that value. The general syntax would be

```
Input.Option1 = 'VALUE1' ;
Input.VALUE1.Val1Opt1 = ...;
Input.VALUE1.Val1Opt2 = ...;
```

This is illustrated as follows:

Table X: Input			
●	.Option1	String	Short description
		'VALUE1'	Description of 'VALUE1'
		'VALUE2'	Description of 'VALUE2'
⊞	.VALUE1	Table Y	Options for 'VALUE1'
⊞	.VALUE2	Table Z	Options for 'VALUE2'

Table Y: Input.VALUE1			
□	.Val1Opt1	String	Description
□	.Val1Opt2	Double	Description

Table Z: Input.VALUE2			
□	.Val2Opt1	String	Description
□	.Val2Opt2	Double	Description

Note: In the sequel, double/doubles mean a real number represented in double precision (resp. a set of such real numbers).

4.1 Create a UQLink Model

Syntax

```
myModel = uq_createModel(Modelopts)
```

Input

The Struct variable `Modelopts` contains the following fields:

Table 2: `Modelopts`

●	`.Command`	String	Sample of the execution command line (see Section 1.4).
●	`.Template`	String or cell	• Name of the input template file(s) that will be used to generate the input files for each realization of the parameters. It shall be renamed by adding another extension to the current input file (See Section 1.4, Section 2.3). • In case of multiple input files, a cell array should be used.
●	`.Output`	Table 3	A structure containing the names of the output file(s) and MATLAB parser. (See Section 1.4, Section 2.4).
□	`.Name`	String	Name of the model.
□	`.Display`	String default: `'standard'`	Level of information displayed during model evaluations.
		`'quiet'`	Minimum display level, displays nothing or very few information.
		`'standard'`	Default display level, shows the most important information.
		`'verbose'`	Maximum display level, shows all the information on runtime, like updates on iterations, etc.
□	`.ExecutablePath`	String default: `''`	Full path of the directory where the executable command is located.
□	`.ExecutionPath`	String default: `''`	Full path of the directory where the input file of the third-party software is located. Execution will be run in this directory.
□	`.Marker`	Cell array of strings default: `{'<','X','>'}`	Marker flag which identifies the parameters to modify.

☐	.Counter	Table 4	A structure containing options defining how the numbering of the created files (input and output) will be carried out.
☐	.Archiving	Table 5	A structure containing information about the archiving of the files generated during the execution of the third-party software.

Table 3: Modelopts.Output

●	.FileName	String or cell	• Name of the output file(s) (where the quantities of interest are stored) generated by the third-party software • In case of multiple output files, a cell array should be used.
●	.Parser	String or function handle	MATLAB function that will retrieve the results from the output file(s).

Table 4: Modelopts.Counter

☐	.Digits	Integer default: 6	Number of digits used in the numbering of the input and output files.
☐	.Offset	Integer default: 0	Offset of the incremental counter for the numbering of the input and output files. For a given value `offset`, the counter starts with `offset+1`

Table 5: Modelopts.Archiving

☐	.Action	String default: 'save'	Specifies how the files generated during execution will be handled.
		'none'	Do nothing.
		'delete'	Delete all the newly generated files.
		'save'	Keep all the files. Subfolders named `UQlinkInput`, `UQlinkOutput` and `UQlinkAux` will be created in a repository to save respectively the input, the output and the auxiliary files (See Section 2.7.5).

☐	`.FolderName`	String default: Name of the MODEL	• Name of the repository where the generated files will be saved. • By default the name of the MODEL is used. All blank spaces are removed. • This option is ignored when `.Archiving.Action` is not set to `'save'`.
☐	`.Zip`	Logical default: true	• A logical to decide whether the archiving repository should be compressed or not. • This option is ignored when `.Archiving.Action` is not set to `'save'`.

Output

After executing the command:

```
myModel = uq_createModel(Modelopts)
```

the object `myModel` is created. It contains the following fields:

Table 6: `myModel`

`.Name`	String	Model name
`.Internal`	Struct	Internal fields used during execution
`.Options`	Struct	Original options used to create the model.

4.2 Evaluate a Model

4.2.1 Basic usage

Syntax

```
Y = uq_evalModel(X)
Y = uq_evalModel(myModel,X)
[Y1,Y2,...] = uq_evalModel(myModel,X)
```

Description

`Y = uq_evalModel(X)` returns the model response of the current MODEL object on the points `X` ($N \times M$ double). `Y` has dimension $N \times O$.

Note: By default, the *last created* model or surrogate model is the currently active model.

`Y = uq_evalModel(myModel,X)` returns the model response of the `MyModel` MODEL object on the points `X`.

`[Y1,Y2,...] = uq_evalModel(myModel,X)` can be used to return an arbitrary number of outputs, if the underlying model supports it.

4.2.2 Advanced usage

4.2.2.1 Recover option

Syntax

```
Y = uq_evalModel(X, 'recover')
Y = uq_evalModel(myModel,X, 'recover')
[Y1,Y2,...] = uq_evalModel(myModel,X, 'recover')
```

Description

The `'recover'` argument can be used to recover previously run but failed simulations (See Section 2.7.8).

4.2.2.2 Recovery source

Syntax

```
Y = uq_evalModel(X, 'recover', 'RecoverySource')
Y = uq_evalModel(myModel,X, 'recover', 'RecoverySource')
[Y1,Y2,...] = uq_evalModel(myModel,X, 'recover', 'RecoverySource')
```

Description

The `'RecoverySource'` argument can be used to specify the `.mat` file in which the previous MODEL evaluations can be found (See Section 2.7.8).

4.2.2.3 Run list

Syntax

```
Y = uq_evalModel(X, 'recover', RunList)
Y = uq_evalModel(myModel,X, 'recover', RunList)
[Y1,Y2,...] = uq_evalModel(myModel,X, 'recover', RunList)
```

Description

The `RunList` argument can be used to specify the indices of the runs that need to be simulated again (See Section 2.7.8).

4.2.2.4 Resume option

Syntax

```
Y = uq_evalModel(X, 'resume')
Y = uq_evalModel(myModel,X, 'resume')
[Y1,Y2,...] = uq_evalModel(myModel,X, 'resume')
```

Description

The `'resume'` argument can be used to resume MODEL evaluations that were stopped before completion (See Section 2.7.9).

Bibliography

Bae, H.-R. and E. Alyanak (2016). Sequential subspace reliability method with univariate revolving integrationn. *AIAA Journal 54*, 2160–2170. 20, 22

Dassault Systèmes (2017). *ABAQUS/Standard User's Manual, Version 6.14*. Simulia. 17

de Rocquigny, E., N. Devictor, and S. Tarantola (2008). *Uncertainty in industrial practice – A guide to quantitative uncertainty management*. John Wiley & Sons. 1

McKenna, F., M. H. Scott, and G. Fenves (2010). Nonlinear finite element analysis software architecture using object composition. *Journal of Computing in Civil Engineering 24*, 95–107. 24

Sudret, B. (2007). Uncertainty propagation and sensitivity analysis in mechanical models - Contributions to structural reliability and stochastic spectral methods. Habilitation thesis, Université Blaise Pascal, Clermont-Ferrand, France. 1

Wei, D. and S. Rahman (2007). Structural reliability analysis by univariate decomposition and numerical integration. *Probabilistic engineering mechanics 12*, 27–38. 17, 20, 22