ETH zürich

# UQLab user manual
# The Model module

C. Lataniotis, S. Marelli, B. Sudret

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

Risk, Safety & Uncertainty Quantification

**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**

C. Lataniotis, S. Marelli and B. Sudret, UQLab user manual – The Model module, Report UQLab-V1.2-103, Chair of Risk, Safety & Uncertainty Quantification, ETH Zurich, 2019.

**BibTeX entry**

```
@TECHREPORT{UQdoc_12_103,
author = {Lataniotis, C. and Marelli, S. and Sudret, B.},
title = {{UQLab user manual -- The Model module}},
institution = {Chair of Risk, Safety \& Uncertainty Quantification, ETH Zurich},
year = {2019},
note = {Report \# UQLab-V1.2-103}
}
```

# Document Data Sheet

| | |
|---|---|
| Document Ref. | UQLab-V1.2-103 |
| Title: | UQLAB user manual – The Model module |
| Authors: | C. Lataniotis, S. Marelli, B. Sudret<br>Chair of Risk. Safety and Uncertainty Quantification, ETH Zurich, Switzerland |
| Date: | 22/02/2019 |

| Doc. Version | Date | Comments |
|---|---|---|
| V1.2 | 22/02/2019 | UQLAB V1.2 release |
| V1.1 | 05/07/2018 | UQLAB V1.1 release |
| V1.0 | 01/05/2017 | UQLAB V1.0 release<br>• Minor consistency improvements in the reference list |
| V0.9 | 01/07/2015 | Initial release |

## Abstract

Computational models are used nowadays in most fields of natural-, social sciences and engineering. The purpose of the UQLAB platform is to quantify the impact of uncertainties in the model parameters onto the predictions of such models. In this manual, we describe how to define such a *computational model* in the UQLab platform.
Models can be as simple as a Matlab handle function or a Matlab `m-file`. Third party codes can be used by wrapping them up into a `m-file`. Surrogate models (see manual UQLAB User Manual – Polynomial chaos expansions and UQLAB User Manual – Kriging (Gaussian process modelling) ) can be built and used later on as any other models.
The manual consists of three sections, namely a short description of the concept of computational models, a section on usage comprising commented examples and a reference list.

**Keywords:** UQLAB, Computational Models, MODEL Module

# 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).

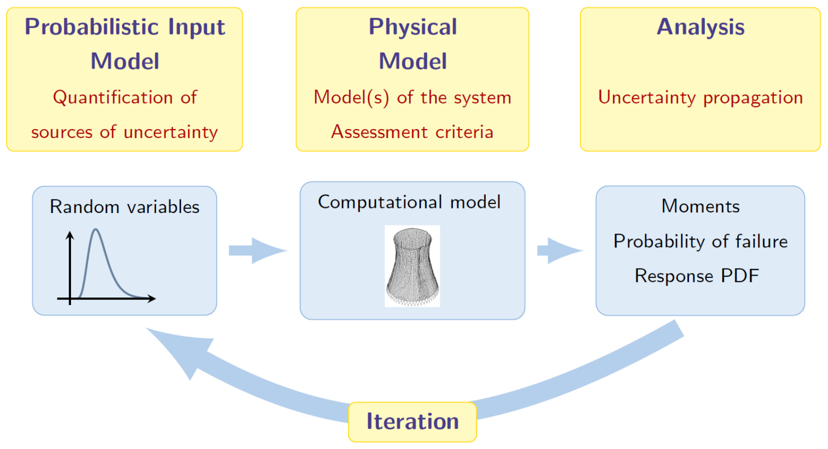

Figure 1: Visual representation of the global theoretical framework for uncertainty quantification developed by Sudret (2007); de Rocquigny et al. (2008), which gives the theoretical foundations to the UQLAB software

This guide provides an in-depth manual on how to connect various types of computational models to the UQLAB software framework.

## 1.2 Formalism

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

$$\boldsymbol{Y} = \mathcal{M}(\boldsymbol{X}) \tag{1.1}$$

where $\boldsymbol{X}$ is a random vector that parametrizes the variability of the input parameters (typically through a joint probability density function PDF) and $\boldsymbol{Y}$ is the vector of model responses. The computational model can be deterministic in the sense that evaluating it repeatedly for a given input realization $\boldsymbol{x}_0$ will give always the same result $\boldsymbol{y}_0 = \mathcal{M}(\boldsymbol{x}_0)$. However, the uncertainty in the input variables causes $\boldsymbol{Y}$ to be a random variable/vector as well. Indeed, one of the main applications of uncertainty quantification is that of propagating the randomness in the inputs $\boldsymbol{X}$ to the outputs $\boldsymbol{Y}$. Commonly used models in advanced application often comprise both random and deterministic parameters. When this is the case, the following notation is introduced to clarify such distinction:

$$\boldsymbol{Y} = \mathcal{M}(\boldsymbol{X}, \boldsymbol{P}) \tag{1.2}$$

where $\boldsymbol{P}$ is a set of deterministic parameters that are used to properly configure a model (e.g. configuration options, fixed values for parametric functions, etc.).

## 1.3 Model examples

As introduced in Section 1.1 a model is a rather abstract concept. In the following a short overview of some of the most common model families in uncertainty quantification is given:

- *Analytic functions*. Any function of the form $\boldsymbol{y} = f(\boldsymbol{x})$ can be considered as a computational model. Mathematical functions are normally known in their closed-forms and they are often used during the development of new algorithms as well-known benchmarks.
- *Numerical models*. The vast majority of physical phenomena cannot be approximated by simple closed-form equations. Advanced discretization techniques (*e.g.*finite difference or finite element schemes) are used to calculate the model response. This type of models can be very demanding computationally.
- *Wrappers to third-party codes*. Sometimes complex simulations (e.g. multiphysics models) rely on the execution of codes that require third party software. The input-output structure of such software may be significantly different than that in UQLAB. Wrappers are small MATLAB codes that "translate" the input-output format of a third party code to and from the input-output format of UQLab.
- *Workflows*. In many industrial applications complex models require composite workflows for proper execution. Such workflows can comprise of several wrappers between different third-party codes, access to storage devices, execution of batch scripts etc. A workflow is a generalization of the concept of wrapper to the general working environment.

The UQLAB MODEL module offers a convenient tool to include the first three types of models (functions, numerical models and wrappers) in an uncertainty quantification analysis.

# Chapter 2

# Usage

## 2.1 Creating models based on analytic functions

MATLAB offers two ways to define analytic functions: m-files and function handles. Both methods are supported in UQLAB. In addition, it is also possible to directly use simple strings to create models. In this section the Ishigami function, a commonly used benchmark in sensitivity analysis is chosen to showcase the available options to create a model in UQLAB:

$$f(\boldsymbol{x}) = \sin(x_1) + a\,\sin^2(x_2) + b\,x_3^4\,\sin(x_1) \tag{2.1}$$

where $a = 7$ and $b = 0.1$ are scalar values. The input vector $\boldsymbol{X}$ comprises 3 components uniformly distributed in the interval $X_i \sim \mathcal{U}(-\pi, \pi)$.

### 2.1.1 Creating a model from an existing m-file

In this section three steps will be covered:

- Create an m-file that implements the Ishigami function
- Create a UQLAB model object based on it
- Use the created object to calculate model responses

To create an Ishigami model named `ishigami_function`, type the following at the MATLAB prompt:

```
edit ishigami_function.m
```

An editor window will open with an empty file. To define the function simply translate Eq. (2.1) in MATLAB syntax as follows:

```
    function Y = ishigami_function(X)
      Y = sin(X(:,1))+7*(sin(X(:,2)).^2)+0.1*(X(:,3).^4).*sin(X(:,1));
```

Note that function is *vectorized*, *i.e.*it allows for the evaluation of $N$ different realizations $\boldsymbol{\mathcal{X}} = \left\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(N)}\right\}$, $\boldsymbol{x}^{(i)} \in \mathbb{R}^M$ in a single call. Within UQLAB the standard convention is used to represent $\boldsymbol{\mathcal{X}}$ as a multi-dimensional matrix `X` of dimension $N \times M$:

```
size(X) = [N, M];
```

Therefore, each row $i$ of the `X` matrix corresponds to a realization $\boldsymbol{x}^{(i)}$, while each column $j$ corresponds to the $j$-th component of each vector $\boldsymbol{x}_j^{(i)}$, or `X(i,j)` $= x_j^{(i)}$. The output matrix of model responses `Y` has the same format, *i.e.*`Y(i,j)` $= y_j^{(i)}$. Therefore, in the case of the scalar-valued Ishigami function, `size(Y) = [N, 1]`.
A MODEL object can be created in UQLAB from the just created function as follows:

```
% Start the framework (if not already started)
uqlab
% Define the model options
modelopts.mFile = 'ishigami_function';
% Create and add the model to UQLab
myModel = uq_createModel(modelopts);
```

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

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

### Vectorization

The default behaviour of UQLAB is to assume that m-files are vectorized. However, it is possible to specify that the provided function is not vectorized, so that UQLAB calculates the model responses automatically on one row of the input matrix `X` at a time. Consider the following file `ishigami_function_nonVec.m` m-file that implements a non-vectorized version of the Ishigami function:

```
function Y = ishigami_function_nonVec(X) % non-vectorized version
  Y = sin(X(1))+7*(sin(X(2)).^2)+0.1*(X(3).^4).*sin(X(1));
```

then the following syntax enables non-vectorized model evaluations in UQLAB:

```
% Define the model options
modelopts.mFile = 'ishigami_function_nonVec';
modelopts.isVectorized = false;
% Create and add the model to UQLab
myModel = uq_createModel(modelopts);
```

Note that, due to the MATLAB programming language design, the performances of non-vectorized functions are greatly reduced (sometimes by orders of magnitude) w.r.t. their vectorized counterparts.

### Passing parameters to an m-file model

In many cases model functions require additional non-random parameters to execute properly, *e.g.*non-random values, flags or even configuration files. It is possible to include non-random parameters in MODEL object with the `.Parameters` structure.
A parametric function in UQLAB is a function with two input parameters of the form `Y = myFunction(X, P)`, where `P` is a structure of parameters. As an example, a parametric

version of the Ishigami function `ishigami_function_parametric.m` could read as follows:

```
function Y = ishigami_function_parametric(X, P)
Y = sin(X(:,1))+P(1)*(sin(X(:,2)).^2)+P(2)*(X(:,3).^4).*sin(X(:,1));
```

To set the parameters to the same values as in the previous examples ($a = 7, b = 0.1$), the following code can be used:

```
  modelopts.mFile = 'ishigami_function_parametric';
  modelopts.Parameters = [7  0.1];
  % Create the module
  myModel = uq_createModel(modelopts);
```

The parameter values can be changed at any time on an existing MODEL object by modifying its `.Parameters` property. As an example, one can set the value $b = 0.05$ in the previous example as:

```
  myModel.Parameters(2) = 0.05;
```

For details on the `.Parameters` structure refer to Section 2.1.4.

### 2.1.2 Creating a model from a function handle

Function handles can also be used to quickly define models in UQLAB. To directly define an Ishigami function-based MODEL object in UQLAB, the following syntax can be used:

```
f = @(X) sin(X(1))+7*(sin(X(2))^2) + 0.01*(X(3)^4) * sin(X(1));
modelopts.mHandle = f;
% Create and add the model to UQLab
myModel = uq_createModel(modelopts);
```

#### Vectorization

By default function-handle-based models in UQLAB are assumed to be *non-vectorized*. However, it is possible to define vectorized handles for improved performance by setting the `.isVectorized` option to `true`:

```
f = @(X)  sin(X(:,1))+7*(sin(X(:,2)).^2)+0.1*(X(:,3).^4).*sin(X(:,1));
modelopts.mHandle = f;
modelopts.isVectorized = true;
myModel = uq_createModel(modelopts);
```

#### Parametric function handles

Parametric handles can also be defined by adding a second argument to the handle definition as well as defining an appropriate `.Parameters` structure:

```
f = @(X,P) sin(X(1)) + P(1)*(sin(X(2))^2) + P(2)*(X(3)^4)*sin(X(1));
modelopts.mHandle = f;
modelopts.Parameters = [7 0.1];
myModel = uq_createModel(modelopts);
```

Model parameters can then be set with the `.Parameters` property:

```
myModel.Parameters(2) = 0.05;
```

For details of the `.Parameters` structure refer to Section 2.1.4.

### 2.1.3 Creating a model from a text string

UQLAB also accepts model definitions through text strings. The syntax is very similar to function handles (Section 2.1.2), but it does not require the user to specify the variable and parameters before the function definition. Function strings can be used to define all types of models: vectorized, non-vectorized and parametric. The convention to write a function string in UQLAB is to indicate the random variables with the letter `X` (**capital** X) and the function parameters with the letter `P` (**capital** P). The non-parametric, non-vectorized version of the Ishigami function can be defined as:

```
modelopts.mString='sin(X(1))+7*(sin(X(2))^2)+0.01*(X(3)^4)*sin(X(1))';
myModel = uq_createModel(modelopts);
```

#### Vectorization

By default function-string-based models in UQLAB are assumed to be non-vectorized. However, it is possible to define vectorized handles for improved performance by setting the `.isVectorized` option to `true`:

```
modelopts.mString=...
    'sin(X(:,1))+7*(sin(X(:,2)).^2)+0.01*(X(:,3).^4).*sin(X(:,1))';
modelopts.isVectorized = true;
myModel = uq_createModel(modelopts);
```

#### Parametric strings

Parametric strings can also be defined by adding the parameters in a variable named `P`, and they can be assigned with the `.Parameters` structure:

```
modelopts.mString=...
'sin(X(:,1)+P(1)*(sin(X(:,2))^2)+P(2)*(X(:,3)^4)*sin(X(:,1))';
modelopts.Parameters = [7  0.1];
myModel = uq_createModel(modelopts);
```

Model parameters can then be changed at any stage with the `.Parameters` property (see Section 2.1.1). To change to $b = 0.05$, one can write, *e.g.*

```
myModel.Parameters(2) = 0.05;
```

For details on the `.Parameters` structure refer to Section 2.1.4.

### 2.1.4 Advanced parameter options

The `.Parameters` option in the previous examples was always a vector of doubles. However, there is no restriction on its format. A common alternative can be a structure with arbitrary fields. As an example, the parametric string-based model example in Section 2.1.3 can equivalently be rewritten in the following form:

```
modelopts.mString=...
'sin(X(:,1))+P.a*(sin(X(:,2))^2)+P.b*(X(:,3)^4)*sin(X(:,1))';
modelopts.Parameters.a = 7;
modelopts.Parameters.b = 0.1;
myModel = uq_createModel(modelopts);
```

Note that there is complete freedom in the form of `.Parameters` option, including, e.g., Cell arrays, MATLAB objects, text strings etc.. In other words, provided a model of the form `Y = myModel(X,P)`, no restrictions on the number or format of the parameters in the structure `P` are given.

## 2.2 Using MATLAB-based numerical models

The second class of models introduced in Section 1.3 is that of MATLAB-based computational models. Typical examples include finite difference (DF) or finite elements (FEM) models. They differ from analytical functions in that they are usually computationally much more expensive, relying on the numerical evaluation of integrals or other time-consuming algorithms. In addition, they are rarely self-contained in single files, but rather they constitute full MATLAB-based software packages.
Thanks to the black-box design of UQLAB, however, such differences are mostly irrelevant to the construction of a numerical-model-based MODEL object. Indeed, the only relevant difference between using a function and a numerical model (assuming it retains the form `Y = myNumericalModel(X,P)`), is that the user must make sure that all the files needed for the model's execution can be found in the current MATLAB path.
To use an existing MATLAB numerical modelling code, it is recommended to use the `mFile` configuration option. Assuming the model executable function is `Y = myNumericalModel(X)`, one can create a MODEL object as follows:

```
modelOpts.mFile = 'myNumericalModel';
myModel = uq_createModel(modelOpts);
```

**Note:** by default UQLAB considers `mFile`-based models as vectorized, hence care should be taken into properly setting the `modelOpts.isVectorized` flag (see Section 2.1.1).

Parameters can be passed normally to the MODEL object generated via the `.Parameters` structure.

## 2.3 Creating wrappers/plugins to third party software

There are several ways to connect external sofware to a MATLAB session. These include special plugins (e.g., COMSOL *Livelink*), interfaces to different languages (e.g., MATLAB

*MEX* compiler), shell/system integration (e.g., MATLAB `system` command) and many others. In this section the guidelines on how to write wrapper codes/plugins that can be used by UQLAB's built-in MODEL module are presented.

### 2.3.1 Understanding wrappers

A *code wrapper*, also known as a plug-in, 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. Therefore, four main points are needed to create a wrapper suitable for UQLAB:

- **The input format of the software that is to be connected to UQLAB:** Most high level modelling software use as input parameters human-readable configuration files as well as a command line to be executed.
- **The output format of the software that is to be connected to UQLAB:** after the computation is finished, the results are normally saved in output files with pre-determined format. Depending on the software, they may be plain text or binary (sometimes even in proprietary format) files, or in rare cases even simple text output printed directly on screen.
- **The input/output format of the UQLAB modelling interface:** thanks to its black-box-centric design, UQLAB expects wrappers to external software to behave as simple functions, with the same input-output structure as any other allowed functions described in Section 2.1.

With this information, it is straightforward to create a wrapper function in MATLAB that performs the following operations:

1. Receive and interpret a random input vector sample and a set of deterministic parameters
2. Create a set of input configuration files/command lines according to the third party software format
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

### 2.3.2 Creating a UQLAB MODEL plugin

The API for UQLAB models is quite simple: a wrapper is a MATLAB function of the form:

```
function Y = myWrapper(X,Parameters)
```

where `X` is a $N \times M$ matrix containing $N$ samples of the $M$ input stochastic parameters, `.Parameters` is a free-form structure containing the static parameters used by the external software (*e.g.*command line options, configuration file names or other configuration options) and `Y` is the $N \times N_{out}$ vector of model responses as calculated by the external software. The actual implementation of the wrapper is a complete black-box to UQLAB. Note that any function (including p-coded and compiled ones) that can be executed from the MATLAB command line with the simple syntax `Y = myWrapper(X,Parameters)`, with `X` and `Y` in the format described before, is an eligible model for UQLAB.

**A simple generic wrapper**

Given the great variety of available modelling software, it is impossible to provide a detailed example of a plugin that would be usable on a wide sample of systems. All the wrappers, however, share a common structure that can be summarized in the following pseudocode:

```
function Y = myWrapper(X,Parameters)
%% 1. retrieve the static configuration parameters
Config1 = Parameters.Config1;
Config2 = Parameters.Config2;
...
%% 2. calculate the model response on each sample
for ii = 1:size(X,1)
  create_ExternalInputFiles(X(ii,:),Config1,Config2,...);
  execute_ExternalCode(...);
  Y(ii,:) = retrieve_Results(...);
end
```

The `create_ExternalInputFiles(X(ii,:),Config1,Config2,...)` function translates the inputs provided by the `X` matrix and `.Parameters` structure into the format required by the external code, one sample at a time.
The `execute_ExternalCode(...)` line contains the necessary commands needed by MATLAB to execute the external code. It may be a simple MATLAB function (*e.g.* from another toolbox) or an external program (*e.g.* executed through MATLAB `system` command).
Finally, the `Y(:,ii) = retrieve_Results(...)` function retrieves the results from the execution of the external program and translates them in the format of the output `Y` matrix. This function can be a simple operation that reformats the output of the external software (*e.g.* if it was a MATLAB script), or a complex function that reads several files and merges their results together.
Note that this example is intended to be used only as a generic guideline.

## 2.4 Models with multiple outputs

It is sometimes the case that computational models can return more than one output at a time ($N_{out} > 1$). This is fully supported and there is no need for any extra configuration options. The vector output responses are expected to be contained in an $N \times N_{out}$ matrix.

# Chapter 3

# 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).

## 3.1 Create a Model

### Syntax

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

### Input

The Struct variable `Modelopts` contains the following fields:

Table 1: `Modelopts`

| | | | |
|---|---|---|---|
| ⊕ | `.mFile` | String | File name of the model function (see Section 2.1.1) |
| ⊕ | `.mHandle` | Function handle | Function handle to be used as model (see Section 2.1.2) |
| ⊕ | `.mString` | String | String containing the model expression (see Section 2.1.3) |
| ☐ | `.Parameters` | Any data type | Non-random model parameters (see Section 2.1.1) |
| ☐ | `.isVectorized` | Logical<br>default:<br>true if m-file,<br>false if mString or mHandle | Set to true if the model function supports vectorized input and to false otherwise |

### Output

After executing the command:

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

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

Table 2: `myModel`

| | | |
|---|---|---|
| `.Name` | String | Model name |
| `.Internal` | Table 3 | Internal fields used during execution |
| `.Options` | Structure | Original options used to create the model |
| `.Parameters` | Any data type | Parameters structure |
| `.isVectorized` | Logical | The parameters that are used for the model evaluations. |
| `.mFile` | String | File name of the model function (see Section 2.1.1) |
| `.mHandle` | Function handle | Function handle to be used as model (see Section 2.1.2) |
| `.mString` | String | String defining the model expression (see Section 2.1.3) |

Table 3: `myModel.Internal`

| | | |
|---|---|---|
| `.Runtime` | Structure | Parameters used by the model during model evaluations |
| `.Location` | String | Path of the `m-file` that contains the model or wrapper. Empty for handle- and string-based MODEL objects |
| `.fHandle` | Function handle | Handle used internally for model evaluations |

## 3.2 Evaluate a Model

### 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 $N \times N_{out}$.

**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.

# Bibliography

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

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