ETH zürich

UQLib
User Manual

M. Moustapha, C. Lataniotis, P. Wiederkehr, P.-R. Wagner, D. Wicaksono, 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, C. Lataniotis, P. Wiederkehr, P.-R. Wagner, D. Wicaksono, S. Marelli, B. Sudret, UQLib – User manual, Report UQLab-V2.0-201, Chair of Risk, Safety and Uncertainty Quantification, ETH Zurich, Switzerland, 2022

BibTeX entry

```
@TechReport{UQdoc_20_201,
author = {Moustapha, M. and Lataniotis, C. and Wiederkehr, P. and and Wagner, P.-R.
and Wicaksono, D. and Marelli, S. and Sudret, B.},
title = {{UQLib user manual}},
institution = {Chair of Risk, Safety and Uncertainty Quantification, ETH Zurich,
Switzerland},
year = {2022},
note = {Report UQLab-V2.0-201}
}
```

Document Data Sheet

Document Ref.	UQLAB-V2.0-201
Title:	UQLIB – User manual
Authors:	M. Moustapha, C. Lataniotis, P. Wiederkehr, P.-R. Wagner, D. Wicaksono, S. Marelli, B. Sudret Chair of Risk, Safety and Uncertainty Quantification, ETH Zurich, Switzerland
Date:	01/02/2022

Doc. Version	Date	Comments
V2.0	01/02/2022	UQLAB V2.0 release
V1.4	01/02/2021	UQLAB V1.4 release • Added the `uq_map` function
V1.3	19/09/2019	UQLAB V1.3 release • Added the Graphics library
V1.2	22/02/2019	First release of UQLIB

Abstract

UQLIB is a collection of general-purpose open-source MATLAB libraries that are useful in the context of uncertainty quantification. These functions are currently used across the scientific modules of UQLAB, but they are designed for generic use.

This user manual serves as a reference documentation for all the relevant functions of UQLIB. The manual includes the algorithm and explanation behind each library, its syntax, input and output, and at least one example demonstrating its usage.

In the current release, UQLIB includes the following libraries:

- Differentiation
- Optimization
- Kernels
- Input/output processing
- Graphics

Keywords: UQLAB, Differentiation, Optimization, Kernels, Input/output processing, Graphics

Contents

Introduction

UQLIB is a collection of general-purpose open-source MATLAB functions that are useful in computational science and engineering, particularly in the context of uncertainty quantification (UQ). The functions were originally created during the development of the UQLAB scientific modules, but they are designed to be usable standalone.

UQLIB libraries cover a wide range of computational goals, from optimization to efficient kernel evaluation. In contrast to most other UQLAB user manuals, this manual is not intended to be an introduction to the theory behind a particular problem and its possible solutions. It is conceived instead as a detailed reference guide to deploy the provided functions outside of the UQLAB environment.

1 Organization of the library

UQLIB is organized into different independent libraries. The functions within a library share similar computational goals or objectives. The following sections summarize each of the libraries.

1.1 Differentiation library

Differentiation of mathematical functions is related to efficiently compute gradients. The UQLIB function `uq_gradient` approximates the first-order derivative (gradient) of a multi-dimensional function at multiple points.

1.2 Optimization library

Optimization is the process of finding the minimum or maximum of a multi-dimensional function. In general, optimization algorithms can be split into *local* and *global* optimizers. The former relies on local information, *e.g.*, gradients, to iteratively solve the optimization problem, while the latter has a larger scope in exploring the entire search space.

The optimization library of UQLIB comprises global optimization algorithms for the solution of continuous single-objective problems. The following algorithms currently are available:

- Grid-search optimization (`uq_gso`)
- Cross-entropy optimization (`uq_ceo`)

- Covariance matrix adaptation–evolution strategy (CMA-ES) optimization (uq_cmaes)
- (1+1)-Covariance matrix adaptation–evolution strategy ((1+1)-CMA-ES) optimization (uq_1p1cmaes)
- Constrained (1+1)-Covariance matrix adaptation–evolution strategy (Constrained (1+1)-CMA-ES) optimization (uq_c1p1cmaes)

Each of these implementations can handle bound constraints, but only the variant of CMA-ES (*i.e.*, the last algorithm) can handle non-linear constraints.

1.3 Kernel library

Multi-dimensional kernel functions are useful in a variety of applications such as function interpolation, Gaussian process modeling, representation of random fields, etc. An arbitrary function is generally not a valid kernel as it has to fulfill the so-called *Mercer's conditions* (Cherkassky and Mulier, 2007). Furthermore, kernel functions also feature some parameters that shall be tuned according to a particular application.

The function uq_eval_Kernel computes the kernel matrix of two input matrices for a specified kernel function. The library supports popular stationary and non-stationary kernel functions, as well as custom user-defined kernels.

1.4 Input/Output processing

UQLIB includes miscellaneous functions to assist in the processing of the input and output of an uncertainty quantification analysis tasks using UQLab.

1.4.1 Subsampling

Consider a large sample set $\mathcal{X} = \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(N)}\}$ where each sample point is an M-dimensional vector. In certain contexts, such as metamodeling (Marelli et al., 2021; Lataniotis et al., 2021), having a large number of sample points N leads to high-computational costs or even renders the calculation intractable. *Subsampling* refers to the process of reducing the number of sample points in a way that some of their statistical properties are retained.

In UQLIB, the functions uq_subsample_random and uq_subsample_kmeans create a subsample from a full sample set based on simple random sampling and k-means clustering, respectively.

1.5 Graphics library

This library comprises functions that facilitate the creation of plots with a unified look by allowing to set defaults for selected properties (*e.g.*, font size, grids, color order). Many of the plotting functions wrap the respective MATLAB function and sets the UQLAB default formatting style. Consequently, all input arguments valid for the corresponding MATLAB function are also valid for these functions.

2 Organization of the function documentation

This user manual contains a concise reference for each UQLIB library, as well as at least one example for each function that demonstrates its usage.

Each function in the library is documented according to the following structure:

- **Objective** briefly states the purpose of the function;
- **Algorithm** or **Description** presents the algorithm (if any) or the description of the function as well as provides important references;
- **Syntax** lists all the different possible function calls, followed by brief description for each of the different calls;
- **Examples** gives at least one application of the function showing its input and output;
- **Input** provides exhaustive lists of inputs of the function, including their names, data types, dimension, and short descriptions;
- **Output** provides exhaustive list of outputs from the function, including their names, data types, dimensions, and short descriptions;
- **Notes** gives additional important details and remarks about the function if any, ranging from further detail on the implementation to possible dependencies. If there is no additional remark, this section is excluded.

The input and output sections are presented using a series of tables. The instruction on how to read such tables are given in the following section.

3 How to read the input/output table

A series of tables are used to describe all the inputs and outputs of a given function. Each table commonly contains the name, data types, dimensions (when applicable), and short descriptions.

3.1 Input table

The main inputs of a function is presented in a 4-column table illustrated below, for a function called `uq_foo` having several input arguments: `input1`, `input2`, `options`, and `Name-Value` pairs.

Table 1: `uq_foo(input1, input2, options, Name, Value)`			
●	`input1`	$N \times M$ Double	First input.
●	`input2`	Double	Second input.
			Continued on next page

Table 1–continued from previous page

☐	`options`	Structure, see Table 2	Additional options of the function, as structure.
☐	`Name, Value`	Name-value pairs, see Table 4	Additional options of the function, as name-value pairs.

The first column in the above table indicates whether a given input argument is mandatory, optional, mutually exclusive, etc. A comprehensive list of the symbols and their meaning are 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)

The other three columns in Table 1 correspond to the *name*, *data type*, and *description* of the input argument. When applicable, the dimension of an input argument is given explicitly.

3.2 Structure inputs and outputs

MATLAB structures play an important role in the user interface of UQLAB and therefore UQLIB. They offer a natural way to semantically group configuration options and output quantities. All the field names of a given input or output structure are listed in a separate 3-column table. This is illustrated below for the `options` structure appeared in Table 1.

Table 2: `uq_foo(..., OPTIONS)`

`.Field1`	String default: `'default_string'`	Description of `Field1`.
`.Field2`	Double default: 0.5	Description of `Field2`.
`.Field3`	Logical default: `false`	Description of `Field3`.
`.Field4`	Structure, see Table 3	Description of `Field4`.

The first column in the above table corresponds to the name of the field. Notice that a field of a structure can be identified by the dot notation, *i.e.*, the name is prefixed by a period. The second column corresponds to the data type of the field. When applicable, the dimension and the default value are also given. Finally, the last column corresponds to the short description of the field.

Due to the complexity of the algorithms implemented, it is not uncommon to employ nested structures to fine tune inputs or present more complex outputs. In that case, the fields for each nested input/output structure are elaborated using another 3-column table illustrated below for the `.Field4` structure in Table 2.

Table 3: `options.Field4`		
`.NestedField1`	Double	Description of `NestedField1`.
`.NestedField2`	Integer	Description of `NestedField2`.

3.3 Name-value pair inputs

Another approach to pass options to a function is by specifying the so-called name-value pairs. Using this approach, an optional argument is passed to a function by specifying the *name* of the argument as a string and followed immediately by the value for that particular argument. Several UQLIB functions use name-value pairs to specify optional arguments. All the available argument names and the corresponding valid values of a function are listed in a separate 3-column table as illustrated below.

Table 4: `uq_foo(..., NAME, VALUE)`		
`'NamedArgument1'`	String default: `'Value1'`	Description of the argument `'NamedArgument1'`.
	`'Value1'`	Description of the value `'Value1'`.
	`'Value2'`	Description of the value `'Value2'`.
	`'Value3'`	Description of the value `'Value3'`.
`'NamedArgument2'`	Integer default: 2	Description of the argument `'NamedArgument2'`.

The first column in the above table corresponds to the names of the arguments. Notice that a named argument is always specified as a string. If only a limited selection of values of an argument is possible, these values are listed in the second column of the table as illustrated above for the named argument `'NamedArgument1'`.

3.4 Output table

Finally, UQLIB functions often results in more than a single output. All the outputs of a function are presented in a table similar to the ones shown previously and now illustrated in Table 5. When applicable, the dimension of an output is given in the second column of the table.

Table 5: `[output1,output2,output3] = uq_foo(...)`		
`output1`	Vector Double	Description for `output1`.
`output2`	Matrix Integer	Description for `output2`.
`output3`	Structure	Description for `output3`.

As mentioned in Section 3.2, structures can become outputs of a function. They can also be further nested. The documentation for such outputs is given in separate tables similar to Table 2 and Table 3.

4 Notes on usage

All functions of UQLIB automatically becomes available in the current MATLAB environment upon the launch of UQLAB. These functions, as other UQLAB functions, begin with the prefix `uq_`. Help can be accessed from within MATLAB using either the command `help` or `doc` followed by the name of the function.

Most of the UQLIB functions are, however, self-contained and can be used independently from UQLAB. Dependencies, if any, are noted in the respective **Notes** section of each function documentation.

The source code for the UQLIB functions are available in the `lib` folder inside the main UQLAB installation folder. The subfolders within the `lib` folder are organized according to UQLIB libraries. To get the exact location of a given function within these subfolders, use the command `which` followed by the name of the function in the MATLAB command window.

`uq_gradient` – First-order numerical differentiation

1 Objective

Compute the gradient of a multi-dimensional function at given points.

2 Algorithm

The gradient of a multi-dimensional scalar-valued function $f(\boldsymbol{x}) = f(x_1, x_2, \ldots, x_M)$ is a vector that consists of the partial first-order derivatives of $f(\boldsymbol{x})$ with respect to each dimension:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_i}, \ldots, \frac{\partial f}{\partial x_M}\right)^T. \tag{1}$$

2.1 Finite difference

The basis of numerical approximation for derivatives is Taylor expansion. The function f is expanded using a Taylor expansion for $x_i + h$ while keeping the other dimensions $\boldsymbol{x}_{\sim i}$ constant

$$f\left(x_i + h, \boldsymbol{x}_{\sim i}\right) = f(\boldsymbol{x}) + h f_{x_i} + \frac{1}{2} h^2 f_{x_i, x_i} + \ldots = f(\boldsymbol{x}) + h f_{x_i} + O(h^2), \tag{2}$$

where h is the so-called *step size*, f_{x_i} is the first-order derivative with respect to dimension x_i, O is the higher-order terms, and $O(h^2)$ indicates that the lowest order of these terms is 2. By neglecting higher-order terms and solving for f_{x_i} yields the *finite difference* approximation of the derivative:

$$f_{x_i} \approx \frac{f\left(x_i + h, \boldsymbol{x}_{\sim i}\right) - f(\boldsymbol{x})}{h}. \tag{3}$$

In particular, the above formulation is called the *forward difference* approximation. Truncating the higher order terms in Eq. (3) results in a *truncation error* of order 1.

Eq. (3) can be reformulated if the function f in Eq. (2) is expanded for $x_i - h$. In other words,

$$f\left(x_i - h, \boldsymbol{x}_{\sim i}\right) = f(\boldsymbol{x}) - h f_{x_i} + \frac{1}{2} h^2 f_{x_i, x_i} + \ldots = f(\boldsymbol{x}) - h f_{x_i} + O(h^2). \tag{4}$$

Following the same procedure results in the *backward difference* approximation of the derivative:

$$f_{x_i} \approx \frac{f(\boldsymbol{x}) - f(x_i + h, \boldsymbol{x}_{\sim i})}{h}, \tag{5}$$

in which the order of the truncation error remains 1.

The third approximation results from combining Eq. (4) and Eq. (2). Rearranging and solving for f_{x_i} results in the *centered difference* approximation of the derivative:

$$f_{x_i} \approx \frac{f(x_i + h/2, \boldsymbol{x}_{\sim i}) - f(x_i - h/2, \boldsymbol{x}_{\sim i})}{h}. \tag{6}$$

In this formulation, the order of the truncation error is 2, hence it is more accurate. However, it requires one extra function evaluation per input dimension with respect to the forward and backward differences. Details on the derivation as well as error analysis can be found in Chapra and Canale (2015).

2.2 Methods

`uq_gradient` offers all three methods to approximate the gradient of a function at a given point. The cost of the approximation in terms of the function evaluations N_T is $N_T = N \times (M+1)$ for the forward and backward methods and $N_T = N \times 2M$ for the centered method; where N and M are the number of points and input dimensions, respectively.

Figure 1 illustrates the approximation of the gradient for the function $f(x) = \sin(x)$ at $\boldsymbol{x} = (2.4\pi, \sin(2.4\pi))^T$ by the three methods, assuming a fixed step size of $h = 0.5$. As can be seen the resulting gradient depends on the method.

2.3 Step size h

Choosing the proper value for the step size h is important for numerical accuracy. A large value of h can result in a worse gradient approximation (Figure 1). On the other hand, a very small value of the step size can result in a very small difference between $f(\boldsymbol{x})$ and $f(x_i \pm h, \boldsymbol{x}_{\sim i})$ that can numerically be indiscernible due to the finite precision of floating point operations. In other words, the so-called *round-off* error will start to dominate the approximation. By default, `uq_gradient` uses a fixed $h = 10^{-3}$ for each dimension.

2.4 Vector-valued function

`uq_gradient` supports functions with multiple outputs (*i.e.*, vector-valued functions). In this case, the approximation of the gradient is carried out for each output separately.

3 Syntax

```
G = uq_gradient(X, FUN)
G = uq_gradient(X, FUN, GradientMethod)
G = uq_gradient(X, FUN, GradientMethod, FDStep)
G = uq_gradient(X, FUN, GradientMethod, FDStep, GivenH)
G = uq_gradient(X, FUN, GradientMethod, FDStep, GivenH, KnownX)
G = uq_gradient(X, FUN, GradientMethod, FDStep, GivenH, KnownX,...
```

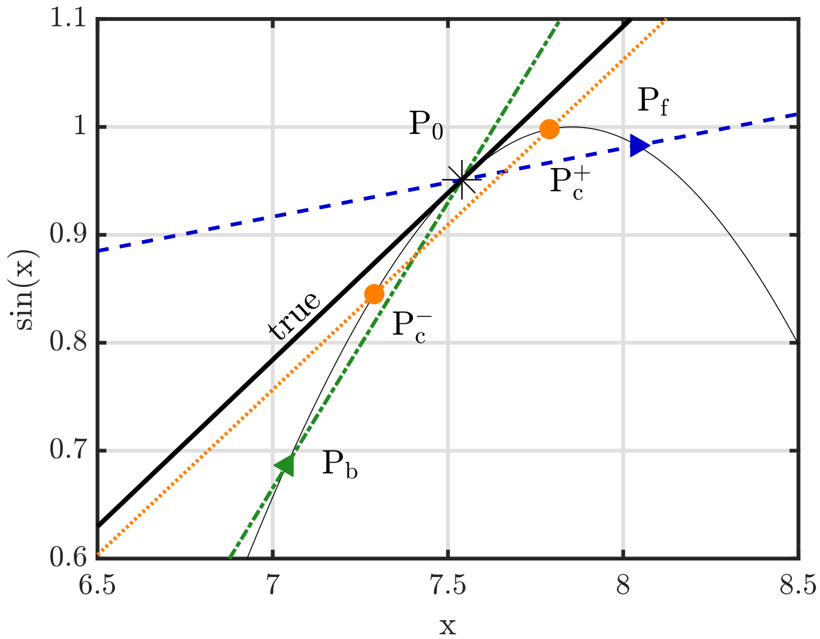


Figure 1: The three methods to estimate the gradient of $f = \sin(x)$ around $x_0 = 2.4\pi$ with step size $h = 0.5$. The forward method approximates the gradient using $P_0 = f(x)$ and $P_f = f(x_0 + h)$ (blue dashed line). The backward method approximates the gradient using P_0 and $P_b = f(x_0 - h)$ (red dash-dot line). Finally, the centered method uses $P_c^- = f(x_0 - 0.5h)$, $P_c^+ = f(x_0 + 0.5h)$ (green dotted line). The black solid line is the true gradient.

```
                    Marginals)
[G,M_X] = uq_gradient(...)
[G,M_X,Cost] = uq_gradient(...)
[G,M_X,Cost,ExpDesign] = uq_gradient(...)
```

`G = uq_gradient(X, FUN)` returns the gradient `G` of the function `FUN` evaluated at the points `X` given as $(N \times M)$ matrix, where N is the number of points and M is the number of input dimensions. It uses the `'forward'` method and a step size of 10^{-3}.

`G = uq_gradient(X, FUN, GradientMethod)` uses the approximation method specified in `GradientMethod` (see Table 1).

`G = uq_gradient(X, FUN, GradientMethod, FDStep)` allows selecting the type of the step size type by specifying `FDStep` (see Table 1).

`G = uq_gradient(X, FUN, GradientMethod, FDStep, GivenH)` allows for adjusting the step size by specifying `GivenH`. The specific effect of `GivenH` on the step size depends on the selected `FDStep` (see Table 1).

`G = uq_gradient(X, FUN, GradientMethod, FDStep, GivenH, KnownX)` uses `KnownX`, a set of precalculated values of `FUN` at `X`, instead of evaluating the function on `X` within the code. If `KnownX` is provided, the cost is reduced by N.

`G = uq_gradient(X, FUN, GradientMethod, FDStep, GivenH, KnownX, Marginals)` uses the standard deviations of input dimensions stored in the structure `Marginals`. `Marginals` is part of a UQLAB INPUT object.

`[G,M_X] = uq_gradient(...)` additionally returns the values of `FUN` at the points `X`.

`[G,M_X,Cost] = uq_gradient(...)` additionally returns the `Cost` of the approximation in terms of the total number of function evaluations.

`[G,M_X,Cost,ExpDesign] = uq_gradient(...)` additionally returns a $1 \times N$ structure array containing the experimental designs used in the approximation of the gradient vector at each given point in `X`.

4 Examples

4.1 Approximate the gradient at different points

Approximate the gradient vector of the function:

$$f(\boldsymbol{x}) = 5 + 2x_1^2 + 3x_2^3 \tag{7}$$

at the points $\boldsymbol{x}^{(1)} = (3, 0.5)$ and $\boldsymbol{x}^{(2)} = (0.5, 1)$. The analytical solution for the gradient at those points are:

$$\nabla f_{|\boldsymbol{x}} = \begin{pmatrix} \nabla f_{|\boldsymbol{x}^{(1)}}{}^T \\ \nabla f_{|\boldsymbol{x}^{(2)}}{}^T \end{pmatrix} = \begin{pmatrix} 12 & 2.25 \\ 2 & 9 \end{pmatrix}$$

The following code approximates the gradient vectors with minimum number of inputs given by the user:

```
fun = @(X) 5 + 2*X(:,1).^2 + 3*X(:,2).^3;
X = [3 0.5; 0.5 1];
G = uq_gradient(X,fun)
```

The code produces:

```
G =

12.0020    2.2545
 2.0020    9.0090
```

in which each row of the output is the gradient vector approximation at a given point.

4.2 Approximate the gradient of a vector-valued function

Approximate the gradient vector of the vector-valued function:

$$\boldsymbol{f}(\boldsymbol{x}) = \left(x_1^3 + x_2^2, \quad \tfrac{2}{3}x_2^{3/2}, \quad 25 + 0.5x_1 + 10x_2, \quad x_1x_2^2\right)^T$$

at the points $\boldsymbol{x}^{(1)} = (3, 4)$ and $\boldsymbol{x}^{(2)} = (3.5, 9)$. The analytical solution for the gradient at those points are defined per output component:

$$\nabla f_{1|\boldsymbol{x}} = \begin{pmatrix} {\nabla f_{1|\boldsymbol{x}^{(1)}}}^T \\ {\nabla f_{1|\boldsymbol{x}^{(2)}}}^T \end{pmatrix} = \begin{pmatrix} 27 & 8 \\ 36.75 & 18 \end{pmatrix} \quad \nabla f_{2|\boldsymbol{x}} = \begin{pmatrix} {\nabla f_{2|\boldsymbol{x}^{(1)}}}^T \\ {\nabla f_{2|\boldsymbol{x}^{(2)}}}^T \end{pmatrix} = \begin{pmatrix} 0 & 2 \\ 0 & 3 \end{pmatrix}$$

$$\nabla f_{3|\boldsymbol{x}} = \begin{pmatrix} {\nabla f_{3|\boldsymbol{x}^{(1)}}}^T \\ {\nabla f_{3|\boldsymbol{x}^{(2)}}}^T \end{pmatrix} = \begin{pmatrix} 0.5 & 10 \\ 0.5 & 10 \end{pmatrix} \quad \nabla f_{4|\boldsymbol{x}} = \begin{pmatrix} {\nabla f_{4|\boldsymbol{x}^{(1)}}}^T \\ {\nabla f_{4|\boldsymbol{x}^{(2)}}}^T \end{pmatrix} = \begin{pmatrix} 16 & 24 \\ 81 & 63 \end{pmatrix}$$

The following code approximates the gradient vectors with minimum number of inputs provided by users:

```
fun =  @(X) [X(:,1).^3+X(:,2).^2 2/3.*X(:,2).^(3/2) ...
             25 + 0.5*X(:,1) + 10*X(:,2) X(:,1).*(X:,2).^2];
X = [3 4;3.5 9];
G = uq_gradient(X,fun)
```

The code produces an $N \times M \times N_{\text{out}}$ multi-dimensional array, where $N = 2$, $M = 2$, and $N_{\text{out}} = 4$ are the numbers of input points, input dimensions, and output dimensions, respectively:

```
G(:,:,1) =

27.0090    8.0010
36.7605   18.0010


G(:,:,2) =

0     2.0001
0     3.0001


G(:,:,3) =

0.5000   10.0000
0.5000   10.0000


G(:,:,4) =

16.0000   24.0030
81.0000   63.0035
```

5 Input

Table 1: uq_gradient(X, FUN, GradientMethod, FDStep, GivenH, KnownX, Marginals)

●	X	$N \times M$ Double	Points at which to approximate the gradient.
●	FUN	$1 \times N_{\text{out}}$ Function handle	Vector-valued function for which the gradient is approximated.
□	GradientMethod	String or function handle default: 'forward'	Method for the gradient approximation.
		'forward'	Use the forward method.
		'backward'	Use the backward method.
		'centered'	Use the centered method.
		Function handle	Use a user-specified function handle to evaluate the gradient on X. The custom function only takes X as input.
□	FDStep	String default: 'fixed'	Specifies the step type.
		'fixed'	Use step size h =GivenH×1.
		'relative'	Use step size h =GivenH×σ_i in the direction of X_i. Only available if Marginals (see below) is provided.
□	GivenH	Double default: 0.001	"Step-size ratio". Used to compute the step size, the effect depends on FDStep (see above).
□	KnownX	$N \times N_{\text{out}}$ Double	Allows user to provide precalculated evaluations of FUN(X).
□	Marginals	Structure	Marginals structure of a UQLAB INPUT object. It is used to read the standard deviations if FDStep is set to 'relative'.

6 Output

Table 2: [G,M_X,Cost,ExpDesign] = uq_gradient(...)

G	$N \times M \times N_{\text{out}}$ Double	Gradient approximations at points X for all outputs of FUN.
M_X	$N \times N_{\text{out}}$ Double	Function evaluations FUN(X). Equal to KnownX if provided.

Continued on next page

Table 2–continued from previous page

`Cost`	Scalar Double	Total number of function evaluations done by `uq_gradient`.
`ExpDesign`	$1 \times N$ Structure Array	Experimental designs built at each point in `X`. Each structure contains the field: • `X`, a Matrix Double of input points. • `Y`, a Vector Double of corresponding function values.

7 Notes

- For probabilistic input, `uq_gradient` offers the possibility of using *relative* step size. In this case, the standard deviations of the inputs are multiplied by h to obtain the step size for each input. Probabilistic inputs are passed into `uq_gradient` via a structure (see `Marginals` in Table 1).

uq_gso – Grid-search optimization

1 Objective

Solve the following unconstrained optimization problem:

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x} \in \mathcal{D}_{\boldsymbol{X}}} f(\boldsymbol{x}), \tag{1}$$

where $\boldsymbol{x} \in \mathcal{D}_{\boldsymbol{X}} \subseteq \mathbb{R}^M$ is an M-dimensional vector; $\mathcal{D}_{\boldsymbol{X}} = \prod_{i=1}^M \left[x_i^{\mathrm{lb}}, x_i^{\mathrm{ub}}\right]$ represents the search space, with the lower and upper bounds of the i-th input dimension x_i^{lb} and x_i^{ub}, respectively; $\boldsymbol{x}^*$ is the optimal solution; and f is a scalar-valued objective function.

2 Algorithm

Grid-search optimization is a heuristic algorithm which consists in finding the minimizer of an objective function among a predefined set of candidates. The algorithm is often used for the calibration of hyperparameters in the context of machine learning applications. The basic idea is to generate a grid over the input space, evaluate the objective function on the generated points, and select the minimizer in this set as the approximate solution.

The procedure is as follows:

1. Initialize the algorithm:
   - Define a set of candidate points to be evaluated or simply set the bounds of the search space.
   - Define the options for the optimizer such as the number of discretization points per input dimension $d_i \geq 2,\ i = 1, \ldots, M$.
2. If a grid is not defined, create one:
   - Generate a uniform discretization along each input dimension: $\mathcal{X}_i = \left\{x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(d_i)}\right\}$, where $x_i^{(1)} = x_i^{\mathrm{lb}}$ and $x_i^{(d_i)} = x_i^{\mathrm{ub}}$.
   - Generate the grid by tensorization: $\mathcal{X} = \prod_{i=1}^M \mathcal{X}_i$.
3. Evaluate the objective function at the grid points $f\left(\boldsymbol{x}^{(i)}\right), i = 1, \ldots, N$, where N is the size of the grid.

4. Rank (sort) the grid points in increasing order of the objective function values, *i.e.*, $\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(N)}$ defined such that $f\left(\boldsymbol{x}_{(1)}\right) \leq \ldots \leq f\left(\boldsymbol{x}_{(N)}\right)$.
5. Return the approximate solution: $\boldsymbol{x}^* = \boldsymbol{x}_{(1)}$.

3 Syntax

```
XSTAR = uq_gso(FUN, MYGRID, NVARS)
XSTAR = uq_gso(FUN, MYGRID, NVARS, LB, UB)
XSTAR = uq_gso(FUN, [], NVARS, LB, UB)
XSTAR = uq_gso(FUN, MYGRID, NVARS, LB, UB, OPTIONS)
[XSTAR,FSTAR] = uq_gso(...)
[XSTAR,FSTAR,EXITFLAG] = uq_gso(...)
[XSTAR,FSTAR,EXITFLAG,OUTPUT] = uq_gso(...)
```

`XSTAR = uq_gso(FUN, MYGRID, NVARS)` finds a local minimizer of the function `FUN` using only evaluations at a predefined set of points `MYGRID`. `NVARS` is the input dimension (number of design variables) of FUN.

`XSTAR = uq_gso(FUN, MYGRID, NVARS, LB, UB)` finds a local minimizer of the function `FUN` using only evaluations at a predefined set of points `MYGRID`, and only evaluating data points that are within lower and upper bounds defined by `LB` and `UB`, respectively.

`XSTAR = uq_gso(FUN, [], NVARS, LB, UB)` finds a local minimizer of the function `FUN` using on an automatically generated grid with 5 discretization points along each input dimension, within the lower (LB) and upper (UB) bounds.

`XSTAR = uq_gso(FUN, MYGRID, LB, UB, OPTIONS)` finds a local minimizer of the function `FUN` using only evaluations at a predefined set of points `MYGRID`, with the default optimization options replaced by the values in the `OPTIONS` structure (see Table 2).

`[XSTAR,FSTAR] = uq_gso(...)` additionally returns the value of the objective function at the solution `XSTAR`.

`[XSTAR,FSTAR,EXITFLAG] = uq_gso(...)` additionally returns an exit flag that indicates the exit condition of the algorithm, either an optimal solution is found or all specified points fall outside the bounds (see Table 3).

`[XSTAR,FSTAR,EXITFLAG,OUTPUT] = uq_gso(...)` returns an additional structure with information about the optimization process (see Table 3).

4 Examples

4.1 Minimize Rosenbrock's function

Consider the minimization problem of the Rosenbrock's function:

$$\boldsymbol{x}^* = \underset{\boldsymbol{x}\in[-10,10]^2}{\arg\min}\ 100(x_2 - x_1^2)^2 + (1 - x_1)^2. \tag{2}$$

The minimum of this function is located at $\boldsymbol{x}^* = (1, 1)$ with the minimum value $f^* = 0$.

The following code solves the optimization problem using uq_gso on an automatically generated grid with the default 5 discretization points along each input dimension (see Table 5):

```
fun =  @(X) 100 .* (X(:,2) - X(:,1).^2).^2 + (1 - X(:,1)).^2;
nvars = 2;
lb = [-10 -10];
ub = [10 10];
xstar = uq_gso(fun, [], nvars , lb , ub)
```

The code produces:

```

Local minimum found that satisfies the bound constraints.
obj. value =            1

ans =

0      0
```

This solution differs from the analytical solution, but it is the best one found inside the grid.

4.2 Specify the number of discretization points

By default, the number of discretization points along each input dimension is 5. The total number of points in the generated grid is thus 25. The following code can be used to increase the size of the grid:

```
fun =  @(X) 100 .* (X(:,2) - X(:,1).^2).^2 + (1 - X(:,1)).^2;
nvars = 2;
lb = [-10 -10];
ub = [10 10];
Options.DiscPoints = 30;
xstar = uq_gso(fun,[], nvars, lb, ub, Options)
```

The code produces:

```

Local minimum found that satisfies the bound constraints.
obj. value =     0.128437

xstar =

1.0345    1.0345
```

With 30 discretization points along each input dimension, the total number of points in the generated grid is 900.

5 Input

Table 1: uq_gso(FUN, MYGRID, NVARS, LB, UB, OPTIONS)

●	FUN	Function handle	Objective function to be minimized.
□	MYGRID	$N \times M$ Double	Candidate set for searching the solution.
●	NVARS	INTEGER	Number of variables in the objective function to be optimized (M).
□	LB	Scalar or $1 \times M$ Double default: -Inf	Lower bounds of the search space.
□	UB	Scalar or $1 \times M$ Double default: Inf	Upper bounds of the search space.
□	OPTIONS	Structure, see Table 2	Algorithm-specific options.

Table 2: uq_gso(..., OPTIONS)

.Display	String default: 'final'	Level of output display.
	'none'	Displays no output.
	'iter'	Displays output at each iteration.
	'final'	Displays only the final output.
.isVectorized	Logical default: true	Specifies whether the objective function is vectorized.
	true	Objective function is vectorized.
	false	Objective function is not vectorized.
.DiscPoints	Scalar or $1 \times M$ Double default: 5	Number of discretization points: • The given value must be larger than 1. • When the problem is multi-dimensional and a scalar is given, the value is replicated along all input dimensions.

6 Output

Table 3: `[XSTAR,FSTAR,EXITFLAG,OUTPUT] = uq_gso(...)`

`XSTAR`	$1 \times M$ Double	Optimal solution.
`FSTAR`	Double	Objective function value at the optimal solution.
`EXITFLAG`	Integer	Flag indicating the termination condition of the algorithm
	`1`	Approximate solution found.
	`-1`	None of the user-specified grid points belong to the bounds.
`OUTPUT`	Structure, see Table 4	Diverse information about the optimization process.

Table 4: `[...,OUTPUT] = uq_gso(...)`

`.message`	String	Exit message.
`.funccount`	Integer	Total number of objective function evaluations.
`.History`	Structure, See Table 5	History of all grid points and their corresponding objective function values.

Table 5: `OUTPUT.History`

`.Grid`	$M \times N$ Double	Grid points that have been evaluated in the search for an optimum.
`.Fitness`	$N \times 1$ Double	Objective function values corresponding to the evaluated points.

uq_ceo – Cross-entropy optimization

1 Objective

Solve the following unconstrained optimization problem:

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x} \in \mathcal{D}_{\boldsymbol{X}}} f(\boldsymbol{x}), \tag{1}$$

where $\boldsymbol{x} \in \mathcal{D}_{\boldsymbol{X}} \subseteq \mathbb{R}$ is an M-dimensional vector; $\mathcal{D}_{\boldsymbol{X}} = \prod_{i=1}^{M} \left[x_i^{\mathrm{lb}}, x_i^{\mathrm{ub}}\right]$ represents the search space, with the lower and upper bounds of the i-th input dimension x_i^{lb} and x_i^{ub}, respectively; $\boldsymbol{x}^*$ is the optimal solution; and f is a scalar-valued objective function.

2 Algorithm

The cross-entropy method was originally developed by Rubinstein (1997) for the estimation of the probability of rare events. The method has been adapted by Rubinstein and Davidson (1999) for the solution of continuous and combinatorial optimization problems and consists in sampling iteratively the search space using a parametrized random distribution to converge to the optimal solution. The implementation in UQLIB considers a Gaussian distribution for sampling the candidate solutions.

The algorithm is summarized below following Kroese et al. (2006):

1. Initialize the algorithm:
    - Set the parameters of the initial Gaussian distribution: the mean $\boldsymbol{\mu}_{\boldsymbol{x}}^{[0]}$ and the standard deviation $\boldsymbol{\sigma}_{\boldsymbol{x}}^{[0]}$, which correspond to the *starting point* and initial *global step size* of the algorithm, respectively.
    - Set the internal parameters of the algorithm: the number of points per iteration (or *generation*) N_{pop}; the smoothing parameters α^{CE}, β^{CE}, and q^{CE}; and the number of points in the *elite* sample set of size $N_{\mathrm{el}} = \lfloor \rho \cdot N_{\mathrm{pop}} \rfloor$, where $\lfloor \circ \rfloor$ denotes the floor function and ρ is a coefficient such that $0 < \rho < 1$. The elite sample set corresponds to a subset of the best points with respect to their objective function values.
    - Set $t = 1$, where t is the counter for the algorithm iteration.
2. Sample N_{pop} points $\left\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(N_{\mathrm{pop}})}\right\}$ from a truncated Gaussian distribution

$\mathcal{N}_{[\boldsymbol{x}^{\text{lb}},\boldsymbol{x}^{\text{ub}}]}\left(\boldsymbol{\mu}_{\boldsymbol{x}}^{[t-1]}, \left(\boldsymbol{\sigma}_{\boldsymbol{x}}^{[t-1]}\right)^T \cdot \boldsymbol{I}_{N_{\text{pop}}} \cdot \boldsymbol{\sigma}_{\boldsymbol{x}}^{[t-1]}\right)$, where $\boldsymbol{I}_{N_{\text{pop}}}$ denotes an identity matrix of size $N_{\text{pop}} \times N_{\text{pop}}$.

3. Evaluate the objective function on the sample points $f\left(\boldsymbol{x}^{(i)}\right), i = 1, \ldots, N_{\text{pop}}$.
4. Select N_{el} sample points with the smallest value of the objective function. Those sample points are denoted by $\left\{\boldsymbol{x}^{*(1)}, \ldots, \boldsymbol{x}^{*(N_{\text{el}})}\right\}$.
5. Compute the variable-wise mean and standard deviation of the elite sample:

$$
\begin{aligned}
\widetilde{\boldsymbol{\mu}}_{\boldsymbol{x}}^{[t]} &= \left\{\widetilde{\mu}_{x_1}^{[t]}, \ldots, \widetilde{\mu}_{x_M}^{[t]}\right\}, \quad \widetilde{\mu}_{x_m}^{[t]} = \frac{1}{N_{\text{el}}} \sum_{j=1}^{N_{\text{el}}} x_m^{*(j)} \\
\widetilde{\boldsymbol{\sigma}}_{\boldsymbol{x}}^{[t]} &= \left\{\widetilde{\sigma}_{x_1}^{[t]}, \ldots, \widetilde{\sigma}_{x_M}^{[t]}\right\}, \quad \widetilde{\sigma}_{x_m}^{[t]} = \sqrt{\frac{1}{N_{\text{el}}} \sum_{j=1}^{N_{\text{el}}} \left(x_m^{*(j)} - \widetilde{\mu}_{x_m}^{[t]}\right)^2}.
\end{aligned}
\tag{2}
$$

where $x_m^{*(j)}$ is the m-th component of the j-th elite sample point.

6. Update the parameters of the Gaussian distribution:

$$
\begin{aligned}
\boldsymbol{\mu}_{\boldsymbol{x}}^{[t]} &= \alpha^{\text{CE}} \widetilde{\boldsymbol{\mu}}_{\boldsymbol{x}}^{[t-1]} + (1 - \alpha^{\text{CE}}) \boldsymbol{x}_{\text{best}}, \\
\boldsymbol{\sigma}_{\boldsymbol{x}}^{[t]} &= \beta_t^{\text{CE}} \widetilde{\boldsymbol{\sigma}}_{\boldsymbol{x}}^{[t-1]} + (1 - \beta_t^{\text{CE}}) \boldsymbol{x}_{\text{best}},
\end{aligned}
\tag{3}
$$

where $\boldsymbol{x}_{\text{best}}$ is the best solution found so far and β_t^{CE} is defined as:

$$
\beta_t^{\text{CE}} = \beta^{\text{CE}} + \beta^{\text{CE}} \left(1 - \frac{1}{t}\right)^{q^{\text{CE}}}
\tag{4}
$$

7. If convergence is achieved, stop the algorithm; otherwise increase $t \leftarrow t + 1$ and go to **Step 2**. The following convergence criteria are considered:
   - Maximum number of generations: the algorithm stops if the number of generations (*i.e.*, iterations) reaches a given threshold.
   - Number of stall generations: the algorithm stops if the number of successive iterations without sampling a point that improves the current best solution reaches a given threshold.
   - Number of function evaluations: the algorithm stops if the number of calls to the objective function reaches a given threshold.
   - Stagnation of the objective function: the algorithm stops if the absolute difference between the maximum and minimum of the objective function values over a given number of iterations (*i.e.*, its *range*) is below a given threshold.
   - Stagnation of the solution: the algorithm stops if the possible change in the solution becomes extremely small, *i.e.*, the current Gaussian distribution can only sample points that are extremely close to its mean.
   - Minimum values: the algorithm stops if the value of the objective function falls

below a given threshold.

The algorithm stops when any of these criteria is reached.

3 Syntax

```
XSTAR = uq_ceo(FUN, X0, SIGMA0)
XSTAR = uq_ceo(FUN, X0, SIGMA0, LB, UB)
XSTAR = uq_ceo(FUN, X0, SIGMA0, LB, UB, OPTIONS)
[XSTAR,FSTAR] = uq_ceo(...)
[XSTAR,FSTAR,EXITFLAG] = uq_ceo(...)
[XSTAR,FSTAR,EXITFLAG,OUTPUT] = uq_ceo(...)
```

`XSTAR = uq_ceo(FUN, X0, SIGMA0)` finds a local minimizer of the function `FUN` with `X0` as the starting point and `SIGMA0` as the initial variable-wise standard deviation.

`XSTAR = uq_ceo(FUN, X0, SIGMA0, LB, UB)` defines a set of lower and upper bounds such that `LB(i) <= XSTAR(i) <= UB(i)`. If `LB` and `UB` are finite and `X0 = []` and/or `SIGMA0 = []`, the center of the search space `(LB(i)+UB(i))/2` and $1/6$ of the search space width, *i.e.*, `(UB(i)-LB(i))/6` are used as `X0(i)` and `SIGMA0(i)`, respectively.

`XSTAR = uq_ceo(FUN, X0, SIGMA0, LB, UB, OPTIONS)` minimizes with the default optimization options replaced by the values in the `OPTIONS` structure (see Table 2).

`[XSTAR,FSTAR] = uq_ceo(...)` returns the value of the objective function at the solution `XSTAR`.

`[XSTAR,FSTAR,EXITFLAG] = uq_ceo(...)` returns an exit flag that indicates the termination condition of the algorithm (see Table 3).

`[XSTAR,FSTAR,EXITFLAG,OUTPUT] = uq_ceo(...)` returns a structure with additional information about the optimization process (see Table 3).

4 Examples

4.1 Minimize Rosenbrock's function

Consider the minimization problem of the Rosenbrock's function:

$$\boldsymbol{x}^* = \underset{\boldsymbol{x}\in[-10,10]^2}{\arg\min}\ 100(x_2 - x_1^2)^2 + (1 - x_1)^2. \tag{5}$$

The minimum of this function is located at $\boldsymbol{x}^* = (1, 1)$ with the minimum value $f^* = 0$.

The following code solves the optimization problem using `uq_ceo` by assuming default values for the starting point `X0` and initial global step size `SIGMA0` (see Table 5):

```
rng(100, 'twister') % For reproducible results
fun = @(X) 100 .* (X(:,2) - X(:,1).^2).^2 + (1 - X(:,1)).^2;
lb = [-10 -10];
ub = [10 10];
```

```
xstar = uq_ceo(fun, [], [], lb, ub)
```

The code produces:

```
Maximum number of stall generations (options.nStall) reached
obj. value =     0.0111671

xstar =

0.9071    0.8279
```

4.2 Specify the starting point and initial global step size

By default, the starting point of `uq_ceo` is `X0(i) = (LB(i)+UB(i))/2` while the initial global step size is `SIGMA0(i) = (UB(i)-LB(i))/6`. The following code specifies user-given values for these two parameters:

```
rng(100, 'twister') % For reproducible results
fun = @(X) 100 .* (X(:,2) - X(:,1).^2).^2 + (1 - X(:,1)).^2;
lb = [-10 -10];
ub = [10 10];
x0 = [-5 5];
sigma0 = [2 2];
xstar = uq_ceo(fun, x0, sigma0, lb, ub)
```

which produces:

```
Possible change in X is below options.TolX
obj. value =    5.5885e-24

xstar =

1.0000    1.0000
```

5 Input

Table 1: `uq_ceo(FUN, X0, SIGMA0, LB, UB, OPTIONS)`

●	`FUN`	Function handle	Objective function to be minimized.
⊕	`X0`	$1 \times M$ Double default: `(LB+UB)/2`	Starting point of the optimization algorithm.
⊕	`SIGMA0`	Scalar or $1 \times M$ Double default: `(UB-LB)/6`	Initial global step size.
⊕	`LB`	Scalar or $1 \times M$ Double default: `-Inf`	Lower bounds of the design space.

Continued on next page

Table 1–continued from previous page

⊕	UB	Scalar or $1 \times M$ Double default: Inf	Upper bounds of the design space.
☐	OPTIONS	Structure default: Table 2	Options of the algorithm.

Table 2: `uq_ceo(..., OPTIONS)`

.Display	String default: 'final'	Level of output display.
	'none'	Display no output.
	'iter'	Display output at each iteration.
	'final'	Display only the final output.
.isVectorized	Logical default: true	Specifies whether the objective function is vectorized.
	true	The objective function is vectorized.
	false	The objective function is not vectorized.
.MaxIter	Integer default: $100 \cdot M$	Maximum number of generations.
.nStallMax	Integer default: 50	Maximum number of stall generations.
.MaxFunEval	Positive Integer default: Inf	Maximum number of objective function evaluations.
.TolFun	Double default: 10^{-3}	Tolerance on the objective function: the algorithm stops if the *range* (*i.e.*, the absolute difference between the maximum and minimum values) of the best objective function values over .nStallMax generations is less than or equal to .TolFun.
.TolSigma	Double default: 10^{-3}	Tolerance on the input variable: the algorithm stops when $\boldsymbol{\sigma}_{\boldsymbol{x}}^{[t]}/\boldsymbol{\sigma}_{\boldsymbol{x}}^{[0]}$ is smaller or equal to .TolSigma.
.FvalMin	Double default: -Inf	Minimum cost: the algorithm stops when the objective function is smaller or equal to FvalMin.
.nPop	Integer default: 100	Population size.
.quantElite	Integer default: 0.05	Proportion of nPop that will be used as elite sample (ρ in Section 2).

Continued on next page

Table 2–continued from previous page

.alpha	Double default: 0.4	Smoothing parameter α^{CE} (Eq. (3)).
.beta	Double default: 0.4	Smoothing parameter β^{CE} (Eqs. (3) (4)).
.q	Double default: 10	Smoothing exponent q^{CE} (Eq. (4)).

6 Output

Table 3: `[XSTAR,FSTAR,EXITFLAG,OUTPUT] = uq_ceo(...)`

XSTAR	$1 \times M$ Double	Optimal solution.
FSTAR	Double	Objective function value at the optimal solution.
EXITFLAG	Scalar	Flag indicating the termination condition of the algorithm.
	1	Maximum number of generations reached.
	2	Maximum number of stall generations reached.
	3	Maximum number of function evaluations reached.
	4	Range of FUN over generations is smaller than threshold.
	5	Global step size smaller than threshold.
	6	Minimum objective function value reached.
	<0	No feasible solution was found.
OUTPUT	Structure, see Table 4	Diverse information about the optimization process.

Table 4: `[...,OUTPUT] = uq_ceo(...)`

.message	String	Exit message.
.iterations	Integer	Total number of generations.

Continued on next page

Table 4–continued from previous page

.funccount	Integer	Total number of objective function evaluations.
.History	Structure, see Table 5	History of the optimization process over iterations.
.lastgeneration	Structure, see Table 6	Parameters of the last generation.

Table 5: OUTPUT.History

.Xmean	Matrix Double	History of the mean of the Gaussian distribution at each iteration.
.sigma	Matrix Double	History of the global step size at each iteration.
.Xbest	Matrix Double	History of the best sampled point at each iteration.
.fitbest	Vector Double	History of the best objective function value at each iteration.
.fitmedian	Vector Double	History of the median of the objective function values at each iteration.

Table 6: OUTPUT.lastgeneration

.Xmean	$1 \times M$ Double	Mean of the final Gaussian distribution.
.Xbest	$1 \times M$ Double	Best solution from the last generation.
.bestfitness	Double	Best objective function value from the last generation.

7 Notes

- The default values for X0 and SIGMA0 (see Table 1) are heuristics.
- The last five values in Table 2 are the CEO-specific parameters. The default values selected there are based on a typical use of CEO for optimizing the hyperparameters of Support Vector Machines MODEL in UQLab (Moustapha et al., 2021b,a).

`uq_cmaes` – Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

1 Objective

Solve the following unconstrained optimization problem:

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x} \in \mathcal{D}_{\boldsymbol{X}}} f(\boldsymbol{x}), \tag{1}$$

where $\boldsymbol{x} \in \mathcal{D}_{\boldsymbol{X}} \subseteq \mathbb{R}$ is an M-dimensional vector; $\mathcal{D}_{\boldsymbol{X}} = \prod_{i=1}^{M} \left[x_i^{\mathrm{lb}}, x_i^{\mathrm{ub}}\right]$ represents the search space, with the lower and upper bounds of the i-th input dimension x_i^{lb} and x_i^{ub}, respectively; $\boldsymbol{x}^*$ is the optimal solution; and f is a scalar-valued objective function.

2 Algorithm

The Covariance Matrix Adaptation–Evolution Strategy (CMA-ES) is a derandomized stochastic search algorithm introduced by Hansen and Ostermeier (2001). The basic idea of the algorithm is to sample points in the search space and adapting the sampling mechanism so as to iteratively move towards the optimal solution. In practice, a Gaussian distribution with a given covariance matrix is considered for sampling. The covariance matrix of the distribution is adapted at each iteration, such that the directions that have improved the objective function in the recent past iterations are more likely to be sampled again. The mean is updated considering a subset of the current iteration best samples (a.k.a. *elite* samples) using a *recombination* scheme with predefined and possibly uneven weights.

Many versions of the algorithms exist, mostly with different selection mechanisms. UQLIB provides the so-called (μ, λ)-CMA-ES. In the (μ, λ)-CMA-ES strategy, the candidate solutions of the next generation consist of λ points sampled from a Gaussian distribution considering only the μ best points of the current generation (Hansen and Kern, 2004; Hansen, 2001). The actual implementation is more complex; the user can refer to Hansen (2001) for details.

The important steps of the algorithm are as follows:

1. Initialize the algorithm:
   - Set the parameters of the initial Gaussian distribution: the mean $\boldsymbol{\mu}_{\boldsymbol{x}}^{[0]} \in \mathbb{R}^M$ and the standard deviation $\boldsymbol{\sigma}_{\boldsymbol{x}}^{[0]} \in R_{>0}^M$ which correspond to the algorithm *starting point* and the *step size*, respectively. The parameter $\sigma^{[0]} = \max\left(\boldsymbol{\sigma}_{\boldsymbol{x}}^{[0]}\right)$ corresponds to the *global* step size.
   - Initialize the covariance matrix $\boldsymbol{C}^{[0]} = \boldsymbol{I}_M$, where I_M is an $M \times M$ identity matrix.
   - Set the internal CMA-ES parameters: the recombination scheme (details are given in Table 2) of the weights $\{w_i,\, i = 1, \ldots, \mu\}$ (see Eq. (3)); the initial *evolution paths* $\boldsymbol{p}_s$ and $\boldsymbol{p}_c$; and the coefficients c_σ, d_σ, c_c, c_{cov}, and μ_{cov}.
   - Set $t = 1$, where t is the counter for the algorithm iteration (or so-called *generation* in CMA-ES optimization).

2. Sample λ points $\left\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(i)}, \ldots, \boldsymbol{x}^{(\lambda)}\right\}$ such that

$$\boldsymbol{x}^{(i)} = \boldsymbol{\mu}_{\boldsymbol{x}}^{[t-1]} + \sigma^{[t-1]} \boldsymbol{B}^{[t-1]} \boldsymbol{D}^{[t-1]} \boldsymbol{z}^{(i)}, \tag{2}$$

   where $\boldsymbol{z}^{(i)} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}_M)$; and $\boldsymbol{B}$ and $\boldsymbol{D}$ are obtained from the eigendecomposition of the covariance matrix $\boldsymbol{C}^{[t-1]}$, *i.e.*, $\boldsymbol{C}^{[t-1]} = \left(\boldsymbol{B}^{[t-1]}\right)^T \left(\boldsymbol{D}^{[t-1]}\right)^2 \boldsymbol{B}^{[t-1]}$.

3. Evaluate the objective function at the sample points $f\left(\boldsymbol{x}^{(i)}\right),\ i = 1, \ldots, \lambda$.

4. Select μ sample points with the smallest value of the objective function. Those sample points are denoted by $\left\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\mu)}\right\}$.

5. Update the mean of the Gaussian distribution:

$$\boldsymbol{\mu}_{\boldsymbol{x}}^{[t]} = \sum_{i=1}^{\mu} w_i \boldsymbol{x}^{(i)}, \tag{3}$$

   where w_i are weights whose values depend on the recombination scheme (see `.recombination` in Table 2).

6. Update the global step size:

$$\sigma^{[t]} = \sigma^{[t-1]} \exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{\|\boldsymbol{p}_s^{[t]}\|}{\sqrt{M}\,\left(1 - \frac{1}{4M} + \frac{1}{21M^2}\right)} - 1\right)\right), \tag{4}$$

   where $\|\cdot\|$ denotes the Euclidean norm and

$$\boldsymbol{p}_s^{[t]} = (1 - c_s)\boldsymbol{p}_s^{[t-1]} + \sqrt{c_s\,(2 - c_s)\,\mu_{\text{eff}}}\,\boldsymbol{B}^{[t-1]} \boldsymbol{\mu}_{\boldsymbol{z}}^{[t-1]}, \tag{5}$$

   where $\mu_{\text{eff}} = 1/\sum_{i=1}^{\mu} w_i^2$ is the so-called *variance effective selection mass*; and $\boldsymbol{\mu}_{\boldsymbol{z}}^{[t-1]} = \sum_{i=1}^{\mu} w_i \boldsymbol{z}^{(i)}$.

7. Update the covariance matrix:

$$\begin{aligned} \boldsymbol{C}^{[t]} &= (1 - c_{\mathrm{cov}})\, \boldsymbol{C}^{[t-1]} + \frac{c_{\mathrm{cov}}}{\mu_{\mathrm{cov}}} \boldsymbol{p}_c^{[t]} \left(\boldsymbol{p}_c^{[t]}\right)^T \\ &+ c_{\mathrm{cov}} \left(1 - \frac{1}{\mu_{\mathrm{cov}}}\right) \sum_{i=1}^{\mu} \frac{w_i}{\left(\sigma^{[t-1]}\right)^2} \left(\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_{\boldsymbol{x}}^{[t-1]}\right) \left(\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_{\boldsymbol{x}}^{[t-1]}\right)^T . \end{aligned} \tag{6}$$

where

$$\boldsymbol{p}_c^{[t]} = (1 - c_c)\boldsymbol{p}_c^{[t-1]} + h_\sigma \frac{\sqrt{c_c\,(2 - c_c)\,\mu_{\mathrm{eff}}}}{\sigma^{[t-1]}} \left(\boldsymbol{\mu}_{\boldsymbol{x}}^{[t]} - \boldsymbol{\mu}_{\boldsymbol{x}}^{[t-1]}\right), \tag{7}$$

and h_σ is defined as

$$h_\sigma = \begin{cases} 1, & \text{if } \sqrt{\frac{\|\boldsymbol{p}_s^{[t]}\|}{1-(1-c_\sigma)^{2(g/\lambda)}}} \Big/ \sqrt{M}\left(1 - \frac{1}{4M} + \frac{1}{21M^2}\right) < 1.4 + \frac{2}{M+1} \\ 0, & \text{otherwise} \end{cases}, \tag{8}$$

where g is the current number of objective function evaluations.

8. If convergence is achieved, stop the algorithm; otherwise increase $t \leftarrow t + 1$ and go back to **Step 2**. The following convergence criteria are considered:

   - Number of generations: the algorithm stops if the number of generations (*i.e.*, iterations) reaches a given threshold.
   - Number of stall generations: the algorithm stops if the number of successive iterations without sampling a point that improves the current best solution reaches a given threshold.
   - Number of function evaluations: the algorithm stops if the number of calls to the objective function reaches a given threshold.
   - Stagnation of the cost: the algorithm stops if the absolute difference between the maximum and minimum of the objective function values over a given number of iterations (*i.e.*, its *range*) is below a given threshold.
   - Stagnation of solution: the algorithm stops if the possible change in the solution becomes extremely small, *i.e.*, it falls below a given threshold.

   The algorithm stops when any one of these criteria is reached.

3 Syntax

```
XSTAR = uq_cmaes(FUN, X0, SIGMA0)
XSTAR = uq_cmaes(FUN, X0, SIGMA0, LB, UB)
XSTAR = uq_cmaes(FUN, X0, SIGMA0, LB, UB, OPTIONS)
[XSTAR,FSTAR] = uq_cmaes(...)
[XSTAR,FSTAR,EXITFLAG] = uq_cmaes(...)
[XSTAR,FSTAR,EXITFLAG,OUTPUT] = uq_cmaes(...)
```

`XSTAR = uq_cmaes(FUN, X0, SIGMA0)` finds a local minimizer of the function `FUN` with `X0` as the starting point and `SIGMA0` as the initial coordinate-wise standard deviation.

`XSTAR = uq_cmaes(FUN, X0, SIGMA0, LB, UB)` defines a set of lower and upper bounds such that `LB(i) <= XSTAR(i) <= UB(i)`. If `LB` and `UB` are finite and `X0 = []` and/or `SIGMA0 = []`, the center of the search space `(LB(i)+UB(i))/2` and $1/6$ of the search space width, *i.e.*, `(UB(i)-LB(i))/6` are used as `X0(i)` and `SIGMA0(i)`, respectively.

`XSTAR = uq_cmaes(FUN, X0, SIGMA0, LB, UB, OPTIONS)` minimizes with the default optimization options replaced by the values in the `OPTIONS` structure (see Table 2).

`[XSTAR,FSTAR] = uq_cmaes(...)` additionally returns the value of the objective function `FSTAR` at the solution `XSTAR`.

`[XSTAR,FSTAR,EXITFLAG] = uq_cmaes(...)` additionally returns an exit flag that indicates the termination condition of the algorithm (see Table 4).

`[XSTAR,FSTAR,EXITFLAG,OUTPUT] = uq_cmaes(...)` additionally returns a structure with additional information about the optimization process (see Table 4).

4 Examples

4.1 Minimize Rosenbrock's function

Consider the minimization problem of the Rosenbrock's function:

$$\boldsymbol{x}^* = \underset{\boldsymbol{x}\in[-10,10]^2}{\arg\min}\ 100(x_2 - x_1^2)^2 + (1 - x_1)^2. \tag{9}$$

The minimum of this function is located at $\boldsymbol{x}^* = [1, 1]$ with the minimum value $f^* = 0$.

The following code solves the optimization problem using `uq_cmaes` by assuming default values for the starting point `X0` and initial global step size `SIGMA0` (see Table 5):

```
rng(100, 'twister') % For reproducible results
fun = @(X) 100 .* (X(:,2) - X(:,1).^2).^2 + (1 - X(:,1)).^2;
lb = [-10 -10];
ub = [10 10];
xstar = uq_cmaes(fun, [], [], lb, ub)
```

The code produces:

```

Possible change in X is below options.TolX
obj. value =  1.03161e-22

xstar =

1.0000    1.0000
```

4.2 Specify the starting point and initial global step size

By default, the starting point of the uq_cmaes is X0(i) = (LB(i)+UB(i))/2 while the initial global step size is SIGMA0(i) = (LB(i)+UB(i))/6. The following code specifies user-given values for these two parameters:

```
rng(100, 'twister') % For reproducible results
fun =  @(X) 100 .* (X(:,2) - X(:,1).^2).^2 + (1 - X(:;1)).^2;
lb = [-10 -10];
ub = [10 10];
x0 = [-5 5];
sigma0 = [2 2];
xstar = uq_cmaes(fun, x0, sigma0, lb, ub)
```

The code produces:

```

Possible change in X is below options.TolX
obj. value =    5.5885e-24

xstar =

1.0000    1.0000
```

5 Input

Table 1: uq_cmaes(FUN, X0, SIGMA0, LB, UB, OPTIONS)

●	FUN	Function handle	Objective function to be minimized.
⊕	X0	$1 \times M$ Double default: (LB+UB)/2	Starting point of the algorithm.
⊕	SIGMA0	$1 \times M$ Double default: (UB-LB)/6	Initial global step size.
⊕	LB	Scalar or $1 \times M$ Double default: -Inf	Lower bounds of the design space.
⊕	UB	Scalar or $1 \times M$ Double default: Inf	Upper bounds of the design space.
□	OPTIONS	Structure, see Table 2	Options of the algorithm.

Table 2: uq_cmaes(..., OPTIONS)

.lambda	Positive Integer default: $4 + \lfloor 3 \log M \rfloor$	Population size (λ).
.mu	Positive Integer default: $\lfloor \lambda/2 \rfloor$	Parent numbers (μ).

Continued on next page

Table 2–continued from previous page

`.recombination`	String default: `'superlinear'`	Computation of the weights w_i used for the recombination in Eq. (3). The weights are normalized before recombination: $w_i = \widehat{w}_i / \sum_i^{\mu} \widehat{w}_i$.
	`'equal'`	Assigns same weight to all parents regardless of their rank: $\widehat{w}_i = 1/\mu$.
	`'linear'`	Assigns weights that varies linearly with respect to the parents rank: $\widehat{w}_i = \mu + 1/2 - i$.
	`'superlinear'`	Assigns weights that vary superlinearly with respect to the parents rank: $\widehat{w}_i = \log(\mu + 1/2) - \log(i)$.
`.boundsHandling`	String default: `'resampling'`	Strategy how to handle out of bounds samples.
	`'resampling'`	Resamples out-of-bound sample points.
	`'penalization'`	Projects out-of-bounds sample points and penalize the corresponding objective (fitness) function values.
`.Display`	String default: `'final'`	Level of display.
	`'none'`	Displays no output.
	`'iter'`	Displays output at each iteration.
	`'final'`	Displays only the final output.
`.MaxIter`	Positive integer default: $\lfloor 10^3 \cdot (M+5)^2/\sqrt{\lambda} \rfloor$	Maximum number of generations.
`.nStallMax`	Positive integer default: $\max\left(70, 10 + \lceil 30 \cdot M/\lambda \rceil\right)$	Maximum number of stall generations.
`.TolFun`	Double default: 10^{-12}	Tolerance on the objective function: the algorithm stops if the *range* (*i.e.*, the absolute difference between the maximum and minimum values) of the best objective function values over `.nStallMax` generations is less than or equal to `.TolFun`.
`.TolX`	Double default: $10^{-11} \cdot \max(\boldsymbol{\sigma}_{\boldsymbol{x}}^{[0]})$	Tolerance on the input `x`: the algorithm stops when the global step size is too small to allow sampling far enough from the current minimum.

Continued on next page

Table 2–continued from previous page

`.MaxFunEval`	Positive Integer default: `Inf`	Maximum number of function evaluations.
`.isVectorized`	Logical default: `true`	Specify if the objective function is vectorized.
`.keepCDiagonal`	Integer default: `0`	Specify if the covariance matrix should be kept diagonal.
	≤ 0	Covariance matrix is never kept diagonal.
	1	Covariance matrix is always kept diagonal.
	> 1	Covariance matrix is kept diagonal for the first `.keepCDiagonal` iterations.
`.isActiveCMA`	Logical default: `true`	Specify if the covariance matrix should be actively adapted (updated), when the current path leads repeatedly to unsuccessful sample points (*i.e.*, sample points that do not improve the current best solution).
`.Strategy`	Structure default: Table 3	Internal parameters of the CMA-ES algorithm. They are already optimized. It is strongly advised not to modify them.

Table 3: `OPTIONS.Strategy`

`.cs`	Double default: $\frac{\mu_{\text{eff}}}{M+\mu_{\text{eff}}+5}$	c_σ in Section 2. See Hansen (2001) for details.
`.ds`	Double default: $1+2\cdot\max\left(0,\sqrt{\frac{\mu_{\text{eff}}-1}{M+1}}+c_\sigma\right)$	d_σ in Section 2. See Hansen (2001) for details.
`.cc`	Double default: $\frac{4+\mu_{\text{eff}}/M}{M+4+2\cdot\mu_{\text{eff}}/M}$	c_c in Section 2. See Hansen (2001) for details.
`.c1`	Double default: $\frac{2}{(M+1.3)^2+\mu_{\text{eff}}}$	c_{cov} in Section 2. See Hansen (2001) for details.
`.cmu`	Double default: $\min\left(1-c_{\text{cov}},2\cdot\frac{\mu_{\text{eff}}-2+1/\mu_{\text{eff}}}{(M+2)^2+\mu_{\text{eff}}}\right)$	μ_{cov} in Section 2. See Hansen (2001) for details.

where $\mu_{\text{eff}} = 1/\sum_{i=1}^{\mu} w_i^2$ is the variance effective selection mass.

Note: These five parameters of the CMA-ES algorithm have been fine-tuned according to Hansen and Ostermeier (2001). It is not advised to modify their default values.

6 Output

Table 4: `[XSTAR,FSTAR,EXITFLAG,OUTPUT] = uq_cmaes(...)`

XSTAR	$1 \times M$ Double	Optimal solution.
FSTAR	Double	Value of the objective function at the optimal solution.
EXITFLAG	Integer	Flag indicating the termination condition of the algorithm.
	1	Maximum number of generations is reached.
	2	Maximum number of stall generations is reached.
	3	Maximum number of function evaluations is reached.
	4	Range of FUN over generations is smaller than threshold.
	5	Current global step size is smaller than threshold.
	<0	No feasible solution was found.
OUTPUT	Structure, see Table 5	Diverse information about the optimization process.

Table 5: `[...,OUTPUT] = uq_cmaes(...)`

.message	String	Exit message.
.iterations	Integer	Total number of generations.
.funccount	Integer	Total number of objective function evaluations.
.History	Structure, see Table 6	History of the optimization process over iterations.
.lastgeneration	Structure, see Table 7	Parameters of the last generation.

Table 6: OUTPUT.History		
.Xmean	Matrix Double	History of the mean of the Gaussian distribution at each iteration.
.sigma	Vector Double	History of the global step size at each iteration.
.Xbest	Vector Double	History of the best sampled point at each iteration.
.fitbest	Vector Double	History of the best objective function value at each iteration.
.fitmedian	Double vector	History of the median of the objective function values at each iteration.

Table 7: OUTPUT.lastgeneration		
.Xmean	$1 \times M$ Double	Mean of the final Gaussian distribution
.Xbest	$1 \times M$ Double	Best solution from the last generation
.bestfitness	Vector Double	Best objective function from the last generation.

7 Notes

- uq_cmaes is a simple implementation of CMA-ES and it is suited for solving optimization problems in UQLab. The code has not been optimized yet. For a better performance, refer to the implementation in the CMA-ES website (last accessed: 06/11/2018).
- Some numerical considerations (*e.g.*, conditioning of the covariance matrix) in uq_cmaes are directly taken from the original MATLAB CMA-ES implementation by Hansen (2001) available in the CMAE-ES website (last accessed: 06/11/2018).
- The default values for X0 and SIGMA0 (see Table 1) are heuristics.
- The parameters $\boldsymbol{p}_s$ and $\boldsymbol{p}_c$ are both initialized to $\mathbf{0}_M$, an M-dimensional vector of zero.

`uq_1p1cmaes` – (1+1)-CMAES

1 Objective

Solve the following unconstrained optimization problem:

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x} \in \mathcal{D}_{\boldsymbol{X}}} f(\boldsymbol{x}), \tag{1}$$

where $\boldsymbol{x} \in \mathcal{D}_{\boldsymbol{X}} \subseteq \mathbb{R}$ is an M-dimensional vector; $\mathcal{D}_{\boldsymbol{X}} = \prod_{i=1}^{M} \left[x_i^{\text{lb}}, x_i^{\text{ub}}\right]$ represents the search space, with the lower and upper bounds of the i-th input dimension x_i^{lb} and x_i^{ub}, respectively; $\boldsymbol{x}^*$ is the optimal solution; and f is a scalar-valued objective function.

2 Algorithm

The (1+1)-CMA-ES algorithm is a variant of the covariance matrix adaptation–evolution strategy (CMA-ES) developed by Igel et al. (2006) and Suttorp et al. (2009) (see `uq_cmaes`). In this variant, one parent generates exactly one offspring and the covariance matrix is adapted so as to favor sampling in the directions that improve the objective function.

`uq_1p1cmaes` follows the algorithm developed in Arnold and Hansen (2010) where an active covariance matrix adaptation is proposed, *i.e.*, the covariance matrix is updated considering both successful and unsuccessful trial steps. Moreover, the algorithm directly resorts to an incremental update of the Cholesky decomposition of the covariance matrix. Both aspects make the algorithm efficient.

Details of the implementation are shown in the following (Arnold and Hansen, 2010):

1. Initialize the algorithm:
   - Set the the starting point $\boldsymbol{x}_{\text{best}} = \boldsymbol{x}^{[0]}$ and the global step size σ.
   - Initialize the state parameters of the algorithm: the search path $\boldsymbol{s} \in \mathbb{R}^M$; the success probability estimate $P_{\text{succ}} \in [0, 1]$; the Cholesky factor $\boldsymbol{A}$ from the decomposition of the covariance matrix $\boldsymbol{C}$ (*i.e.*, $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{A}^T$), and its inverse $\boldsymbol{A}_{\text{inv}}$ (both are initialized to be the $M \times M$ identity matrix).
   - Set the internal CMA-ES parameters: d_p, c_p, c_c, c_{cov}^{+}, P_{target}, and P_{thres}.
   - Set $t = 1$, where t is the counter for the algorithm iteration (or so-called *generation* in CMA-ES-based optimization).

2. Generate an *offspring candidate*:

$$\boldsymbol{x}^{[t]} = \boldsymbol{x}_{\text{best}} + \sigma \boldsymbol{A}\boldsymbol{z}, \tag{2}$$

where $\boldsymbol{z} \sim \mathcal{N}_M(\boldsymbol{0}, \boldsymbol{I}_M)$ and $\boldsymbol{I}_M$ is an $M \times M$ identity matrix.

3. Evaluate the objective function at the offspring candidate $f\left(\boldsymbol{x}^{[t]}\right)$:

- If $f\left(\boldsymbol{x}^{[t]}\right) \leq f\left(\boldsymbol{x}_{\text{best}}\right)$:

  (a) Update the current best solution $\boldsymbol{x}_{\text{best}} \leftarrow \boldsymbol{x}^{[t]}$.

  (b) Update the success probability estimate:

$$P_{\text{succ}} \leftarrow (1 - c_p) P_{\text{succ}} + c_p. \tag{3}$$

  (c) Update the search path:

  – If $P_{\text{succ}} < P_{\text{thres}}$:

$$\boldsymbol{s} \leftarrow (1 - c_c)\boldsymbol{s} + \sqrt{c_c(2 - c_c)}\boldsymbol{z} \quad \text{and} \quad \alpha = 1 - c_{\text{cov}}^{+}. \tag{4}$$

  – Otherwise:

$$\boldsymbol{s} \leftarrow (1 - c_c)\boldsymbol{s} \quad \text{and} \quad \alpha = \left(1 - c_{\text{cov}}^{+}\right) + c_{\text{cov}}^{+} \cdot c_c \left(2 - c_c\right). \tag{5}$$

  (d) Set $\boldsymbol{w} = \boldsymbol{A}_{\text{inv}}\boldsymbol{s}$ and update the Cholesky factor and its inverse:

$$\begin{aligned} \boldsymbol{A} &\leftarrow \sqrt{\alpha}\boldsymbol{A} + \frac{\sqrt{\alpha}}{\|\boldsymbol{w}\|^2}\left(\sqrt{1 + \frac{c_{\text{cov}}^{+}}{\alpha}\|\boldsymbol{w}\|^2} - 1\right)\boldsymbol{s}\boldsymbol{w}^T, \\ \boldsymbol{A}_{\text{inv}} &\leftarrow \frac{1}{\sqrt{\alpha}}\boldsymbol{A}_{\text{inv}} - \frac{1}{\sqrt{\alpha}\|\boldsymbol{w}\|^2}\left(1 - \frac{1}{\sqrt{1 + \frac{c_{\text{cov}}^{+}}{\alpha}\|\boldsymbol{w}\|^2}}\right)\boldsymbol{w}\left[\boldsymbol{w}^T\boldsymbol{A}_{\text{inv}}\right], \end{aligned} \tag{6}$$

  where $\|\cdot\|$ denotes the Euclidean norm.

  (e) Go to **Step 4**

- Otherwise:

  (a) Update the success probability estimate:

$$P_{\text{succ}} \leftarrow (1 - c_p) P_{\text{succ}}. \tag{7}$$

  (b) If $\boldsymbol{x}^{[t]}$ is worse than its 5-th-order ancestor, *i.e.*, $f\left(\boldsymbol{x}^{[t]}\right) \geq f\left(\boldsymbol{x}^{[t-4]}\right)$, update $\boldsymbol{A}$ and $\boldsymbol{A}_{\text{inv}}$ according to Eq. (6) while replacing c_{cov}^{+} with c_{cov}^{-} where

$$c_{\text{cov}}^{-} = \min\left(\frac{0.4}{M^{1.6} + 1}, \frac{1}{2\|\boldsymbol{z}\|^2 - 1}\right). \tag{8}$$

  (c) Go to **Step 4**

4. Update the global step size:

$$\sigma \leftarrow \sigma \exp\left(\frac{1}{d_p}\frac{P_{\text{succ}} - P_{\text{target}}}{1 - P_{\text{target}}}\right). \tag{9}$$

5. If convergence is achieved, stop the algorithm; otherwise increase $t \leftarrow t + 1$ and go back to **Step 2**. The following convergence criteria are considered:
   - Maximum number of generations: the algorithm stops if the number of generations (*i.e.*, iterations) reaches a given threshold.
   - Number of stall generations: the algorithm stops if the number of successive iterations without sampling a point that improves the current best solution reaches a given threshold.
   - Stagnation of the cost: the algorithm stops if the absolute difference between the maximum and minimum of the objective function values over a given number of iterations (*i.e.*, its *range*) is below a given threshold.
   - Stagnation of the solution: the algorithm stops if the possible change in the solution becomes extremely small, *i.e.*, the current normal distribution can only sample points that are extremely close to its mean.
   - Number of function evaluations: the algorithm stops if the number of calls to the objective function reaches a given threshold.

   The algorithm stops when any one of these criteria is reached.

3 Syntax

```
XSTAR = uq_1p1cmaes(FUN, X0, SIGMA0)
XSTAR = uq_1p1cmaes(FUN, X0, SIGMA0, LB, UB)
XSTAR = uq_1p1cmaes(FUN, X0, SIGMA0, LB, UB, OPTIONS)
[XSTAR,FSTAR] = uq_1p1cmaes(...)
[XSTAR,FSTAR,EXITFLAG] = uq_1p1cmaes(...)
[XSTAR,FSTAR,EXITFLAG,OUTPUT] = uq_1p1cmaes(...)
```

`XSTAR = uq_1p1cmaes(FUN, X0, SIGMA0)` finds a local minimizer of the function `FUN` with `X0` as the starting point and `SIGMA0` as the initial global step size.

`XSTAR = uq_1p1cmaes(FUN, X0, SIGMA0, LB, UB)` defines a set of lower and upper bounds such that `LB(i) <= XSTAR(i) <= UB(i)`. If `LB(i)` and `UB(i)` are finite and `X0 = []` and/or `SIGMA0 = []`, the center of the search space `(LB(i)+UB(i))/2` and $1/6$ of the search space width, *i.e.*, `(UB(i)-LB(i))/6` are used as `X0(i)` and `SIGMA0(i)`, respectively.

`XSTAR = uq_1p1cmaes(FUN, X0, SIGMA0, LB, UB, OPTIONS)` minimizes with the default optimization options replaced by the values in the `OPTIONS` structure (see Table 2).

`[XSTAR,FSTAR] = uq_1p1cmaes(...)` additionally returns the value of the objective function at the solution XSTAR.

[XSTAR,FSTAR,EXITFLAG] = uq_1p1cmaes(...) additionally returns an exit flag that indicates the termination condition of the algorithm (see Table 4).

[XSTAR,FSTAR,EXITFLAG,OUTPUT] = uq_1p1cmaes(...) additionally returns a structure with additional information about the optimization process (see Table 4).

4 Examples

4.1 Minimize Rosenbrock's function

Consider the minimization problem of the Rosenbrock's function:

$$\boldsymbol{x}^* = \underset{\boldsymbol{x}\in[-10,10]^2}{\arg\min}\ 100(x_2 - x_1^2)^2 + (1 - x_1)^2. \tag{10}$$

The minimum of this function is located at $\boldsymbol{x}^* = (1, 1)$ with the minimum value $f^* = 0$.

The following code solves the optimization problem using uq_1p1cmaes by assuming default values for the starting point X0 and initial global step size SIGMA0 (see Table 5):

```
rng(123,'twister')  % For reproducible results
fun =  @(X) 100 .* (X(:,2) - X(:,1).^2).^2 + (1 - X(:,1)).^2;
lb = [-10 -10];
ub = [10 10];
xstar = uq_1p1cmaes(fun, [], [], lb, ub)
```

The code produces:

```

The relative change of F was below options.TolFun
obj. value =  1.35004e-17

xstar =

1.0000    1.0000
```

4.2 Specify the starting point and initial global step size

By default, the starting point of the uq_1p1cmaes is X0(i) = (LB(i)+UB(i))/2 and the initial global step size is SIGMA0(i) = (LB(i)+UB(i))/6. The following code specifies user-given values for these two parameters:

```
rng(123,'twister')  % For reproducible results
fun =  @(X) 100 .* (X(:,2) - X(:,1).^2).^2 + (1 - X(:,1)).^2;
lb = [-10 -10];
ub = [10 10];
x0 = [-5 5];
sigma0 = [2 2];
xstar = uq_1p1cmaes(fun, x0, sigma0, lb, ub);
```

The code produces:

```
The relative change of F was below options.TolFun
obj. value =  7.47186e-13

xstar =

1.0000    1.0000
```

5 Input

Table 1: uq_1p1cmaes(FUN, X0, SIGMA0, LB, UB, OPTIONS)

●	FUN	Function handle	Objective function to be minimized.
⊕	X0	$1 \times M$ Double default: (LB+UB)/2	Starting point of the optimization algorithm.
⊕	SIGMA0	Scalar or $1 \times M$ Double default: (UB-LB)/6	Initial global step size.
⊕	LB	Scalar or $1 \times M$ Double default: -Inf	Lower bounds of the search space.
⊕	UB	Scalar or $1 \times M$ Double default: Inf	Upper bounds of the search space.
□	OPTIONS	Structure default: Table 2	Options of the algorithm.

Table 2: uq_1p1cmaes(..., OPTIONS)

.Display	String default: 'final'	Level of output display.
	'none'	Displays no output.
	'iter'	Displays output at each iteration.
	'final'	Displays only the final output.
.MaxIter	Double default: $1000\,(M+5)^2$	Maximum number of generations (iterations).
.nStallMax	Positive integer default: 50	Maximum number of stall generations.
.MaxFunEval	Positive Integer default: Inf	Maximum number of function evaluations.

Continued on next page

Table 2–continued from previous page

`.TolFun`	Double default: 10^{-6}	Tolerance on the objective function: the algorithm stops if the *range* (*i.e.*, the absolute difference between the maximum and minimum values) of the best objective function values over `.nStallMax` generations is less than or equal to `.TolFun`.
`.TolSigma`	Double default: $10^{-11} \cdot \boldsymbol{\sigma}_{\boldsymbol{x}}^{[0]}$	Tolerance on the input `x`: the algorithm stops when the current global step size σ is lower than `.TolSigma`.
`.isActiveCMA`	Logical default: `true`	Specify if the covariance matrix should be actively adapted (updated), when the current path leads repeatedly to unsuccessful sample points (*i.e.*, sample points that do not improve the current best solution).
`.Strategy`	Structure default: Table 3	Internal parameters of the (1+1)-CMA-ES algorithm. They are already optimized. It is strongly advised not to modify them.

Table 3: `OPTIONS.Strategy`		
`.dp`	Double default: $1 + M/2$	d_p in Eq. (9), see Suttorp et al. (2009) for details.
`.Ptarget`	Double default: $2/11$	P_{target} in Eq. (9), see Suttorp et al. (2009) for details.
`.cp`	Double default: $1/12$	c_p in Eqs. (3) and (7), see Suttorp et al. (2009) for details.
`.cc`	Double default: $2/(M+2)$	c_c in Eqs. (4) and (5), see Suttorp et al. (2009) for details.
`.ccov`	Double default: $2/(M^2+6)$	c_{cov}^{+} in Eqs. (4 – 6), see Suttorp et al. (2009) for details.
`.Pthres`	Double default: 0.44	P_{thres} in Eq. (4), see Suttorp et al. (2009) for details.

Note: These six parameters have been fine-tuned for the (1+1)-CMA-ES algorithm (Suttorp et al., 2009). It is not advised to modify their default values.

6 Output

Table 4: `[XSTAR,FSTAR,EXITFLAG,OUTPUT] = uq_1p1cmaes(...)`

XSTAR	$1 \times M$ Double	Optimal solution.
FSTAR	Double	Objective function value at the optimal solution.
EXITFLAG	Scalar	Flag indicating the termination condition of the algorithm
	1	Maximum number of generations reached.
	2	Maximum number of stall generations reached.
	3	Maximum number of function evaluations reached.
	4	Range of FUN over generations is smaller than threshold.
	5	Global step size sigma smaller than threshold.
	<0	No feasible solution was found.
OUTPUT	Table 5	Diverse information about the optimization process.

Table 5: `[...,OUTPUT] = uq_1p1cmaes(...)`

.message	String	Exit message.
.iterations	Integer	Total number of iterations.
.funccount	Integer	Total number of objective function evaluations.
.History	Structure, see Table 6	History of the optimization process over iterations.

Table 6: OUTPUT.History		
.x	Matrix Double	History of the sampled points at each iteration.
.fval	Vector Double	History of the sampled objective function values at each iteration.
.sigma	Matrix Double	History of the global step size at each iteration.

7 Notes

- In the (1+1)-CMA-ES algorithm, the covariance matrix $\boldsymbol{C}$ itself is never explicitly computed nor required.
- In uq_1p1cmaes, the initial values for the Cholesky factor $\boldsymbol{A}$ and its inverse $\boldsymbol{A}_{\text{inv}}$ are the identity matrix, which correspond to the unit-variance diagonal covariance matrix.
- The default values for X0 and SIGMA0 (see Table 1) are heuristics.

`uq_c1p1cmaes` – Constrained (1+1)-CMAES

1 Objective

Solve the following constrained optimization problem:

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x} \in \mathcal{D}_{\boldsymbol{X}}} f(\boldsymbol{x}) \quad \text{subject to:} \quad g_k(\boldsymbol{x}) \leq 0, \quad k = 1, \ldots, K, \tag{1}$$

where $\boldsymbol{x} \in \mathcal{D}_{\boldsymbol{X}} \subseteq \mathbb{R}$ is an M-dimensional vector; $\mathcal{D}_{\boldsymbol{X}} = \prod_{i=1}^{M} \left[x_i^{\mathrm{lb}}, x_i^{\mathrm{ub}}\right]$ represents the search space, with the lower and upper bounds of the i-th input dimension x_i^{lb} and x_i^{ub}, respectively; $\boldsymbol{x}^*$ is the optimal solution; f is a scalar-valued objective function; and g_k are $K > 0$ scalar-valued constraint functions that need to be fulfilled.

2 Algorithm

The constrained (1+1)-CMA-ES algorithm is a variant of the (1+1)-CMA-ES (Arnold and Hansen, 2010) algorithm developed by Arnold and Hansen (2012) where a constraint handling scheme is added (see `uq_1p1cmaes`). (1+1)-CMA-ES itself is a variant of the covariance matrix adaptation–evolution scheme (CMA-ES), where one parent generates exactly one offspring and the covariance matrix is adapted to favor sampling in the directions that improve the objective function (see `uq_cmaes`). In the constrained version, the covariance matrix is also updated when unfeasible points are sampled to decrease the probability of sampling again in such directions.

`uq_c1p1cmaes` implementation follows the algorithm developed in Arnold and Hansen (2012) where an active covariance matrix adaptation is proposed, *i.e.*, the covariance matrix is updated considering both successful and unsuccessful trial steps. Moreover, the algorithm directly resorts to an incremental update of the Cholesky decomposition of the covariance matrix. Both aspects make the algorithm efficient.

Details of the implementation are shown in the following (Arnold and Hansen, 2012):

1. Initialize the algorithm:
    - Set the starting point $\boldsymbol{x}_{\text{best}} = \boldsymbol{x}^{[0]}$ and the global step size σ.

- Initialize the state parameters of the algorithm: the search path $\boldsymbol{s} \in \mathbb{R}^M$; the success probability estimate $P_{\text{succ}} \in [0,1]$; the Cholesky factor $\boldsymbol{A}$ from the decomposition of the covariance matrix $\boldsymbol{C}$ (*i.e.*, $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{A}^T$), and its inverse $\boldsymbol{A}_{\text{inv}}$ (both are initialized to be the $M \times M$ identity matrix); and a set of exponentially fading $\boldsymbol{v}_k$, $k = 1, \ldots, K$, initialized to be zeros.
- Set the internal CMA-ES parameters: c, d_p, c_p, c_c, P_{target}, c_{cov}^+, and β.
- Set $t = 1$, where t is the counter for the algorithm iteration (or so-called *generation* in CMA-ES-based optimization).

2. Generate an *offspring candidate*:

$$\boldsymbol{x}^{[t]} = \boldsymbol{x}_{\text{best}} + \sigma \boldsymbol{A}\boldsymbol{z}, \tag{2}$$

where $\boldsymbol{z} \sim \mathcal{N}_M(\boldsymbol{0}, \boldsymbol{I}_M)$ and $\boldsymbol{I}_M$ is an $M \times M$ identity matrix.

3. Evaluate the constraints at the offspring candidate $g_k\left(\boldsymbol{x}^{[t]}\right)$, $k = 1, \ldots, K$.

4. For all $k = 1, \ldots, K$ such that $g_k\left(\boldsymbol{x}^{[t]}\right) > 0$, update v_k:

$$\boldsymbol{v}_k \leftarrow (1-c)\,\boldsymbol{v}_k + c_c \boldsymbol{A}\boldsymbol{z}. \tag{3}$$

5. If $\boldsymbol{x}^{[t]}$ is not feasible (*i.e.*, $\sum_{k=1}^K \mathbb{I}_{\left(g_k\left(\boldsymbol{x}^{[t]}\right)>0\right)} \geq 1$, where $\mathbb{I}_{(\circ)}$ denotes the indicator function defined in Eq. (5)):

   (a) Update the Cholesky factor $\boldsymbol{A}$:

$$\boldsymbol{A} \leftarrow \boldsymbol{A} - \frac{\beta}{\sum_{k=1}^K \mathbb{I}_{\left(g_k\left(\boldsymbol{x}^{[t]}\right)>0\right)}} \sum_{k=1}^K \mathbb{I}_{\left(g_k\left(\boldsymbol{x}^{[t]}\right)>0\right)} \frac{\boldsymbol{v}_k \boldsymbol{w}_k^T}{\boldsymbol{w}_k^T \boldsymbol{w}_k} \tag{4}$$

   where $\boldsymbol{w}_k = \boldsymbol{A}^{-1}\boldsymbol{v}_k$ and $\mathbb{I}_{(\circ)}$ denotes the indicator function defined by:

$$\mathbb{I}_{\left(g_k\left(\boldsymbol{x}^{[t]}>0\right)\right)} = \begin{cases} 1 & \text{if} \quad g_k\left(\boldsymbol{x}^{[t]}\right) > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

   (b) Go back to **Step 2**.

6. Evaluate the objective function at the offspring candidate $f\left(\boldsymbol{x}^{[t]}\right)$.

7. Update the success probability estimate:

$$P_{\text{succ}} \leftarrow (1-c_p)P_{\text{succ}} + c_p \mathbb{I}_{\left(f\left(\boldsymbol{x}^{[t]}\right) \leq f(\boldsymbol{x}_{\text{best}})\right)}. \tag{6}$$

where $\mathbb{I}_{(\circ)}$ denotes the indicator function defined in Eq. (5).

8. Update the global step size:

$$\sigma \leftarrow \sigma \exp\left(\frac{1}{d_p}\frac{P_{\text{succ}} - P_{\text{target}}}{1 - P_{\text{target}}}\right). \tag{7}$$

9. If $f\left(\boldsymbol{x}^{[t]}\right) \leq f\left(\boldsymbol{x}_{\text{best}}\right)$, go to **Step 10**; otherwise go to **Step 14**.

10. Set $\boldsymbol{x}_{\text{best}} \leftarrow \boldsymbol{x}^{[t]}$.

11. Update the search path:

$$\boldsymbol{s} \leftarrow (1-c)\boldsymbol{s} + \sqrt{c(2-c)}\boldsymbol{A}\boldsymbol{z}. \tag{8}$$

12. Set $\boldsymbol{w} = \boldsymbol{A}^{-1}\boldsymbol{s}$ and update the Cholesky factor and its inverse:

$$\begin{aligned} \boldsymbol{A} &\leftarrow a\boldsymbol{A} + b\boldsymbol{s}\boldsymbol{w}^T, \\ \boldsymbol{A}_{\text{inv}} &\leftarrow \frac{1}{a}\boldsymbol{A}_{\text{inv}} - \frac{b}{a^2 + ab\|\boldsymbol{w}\|^2}\boldsymbol{w}\left[\boldsymbol{w}^T\boldsymbol{A}_{\text{inv}}\right], \end{aligned} \tag{9}$$

where

$$\begin{aligned} a &= \sqrt{1 - c_{\text{cov}}^{+}} \\ b &= \frac{\sqrt{1 - c_{\text{cov}}^{+}}}{\|\boldsymbol{w}\|^2}\left(\sqrt{1 + \frac{c_{\text{cov}}^{+}}{1 - c_{\text{cov}}^{+}}\|\boldsymbol{w}\|^2} - 1\right). \end{aligned} \tag{10}$$

where $\|\cdot\|$ denotes the Euclidean norm.

13. Go to **Step 15**.

14. If $\boldsymbol{x}^{[t]}$ is worse than its 5-th-order ancestor, *i.e.*, $f\left(\boldsymbol{x}^{[t]}\right) \geq f\left(\boldsymbol{x}^{[t-4]}\right)$, set $\boldsymbol{w} = \boldsymbol{A}^{-1}\boldsymbol{s}$ and update $\boldsymbol{A}$ according to Eq. (9) with the coefficients a and b computed as follows:

$$\begin{aligned} a &= \sqrt{1 - c_{\text{cov}}^{-}}, \\ b &= \frac{\sqrt{1 - c_{\text{cov}}^{-}}}{\|\boldsymbol{z}\|^2}\left(\sqrt{1 + \frac{c_{\text{cov}}^{-}}{1 - c_{\text{cov}}^{-}}\|\boldsymbol{z}\|^2} - 1\right), \end{aligned} \tag{11}$$

where

$$c_{\text{cov}}^{-} = \min\left(\frac{0.4}{M^{1.6}+1}, \frac{1}{2\|\boldsymbol{z}\|^2 - 1}\right). \tag{12}$$

15. If convergence is achieved, stop the algorithm; otherwise increase $t \leftarrow t+1$ and go back to **Step 2**. The following convergence criteria are considered:
    - Maximum number of generations: the algorithm stops if the number of generations (*i.e.*, iterations) reaches a given threshold.
    - Number of stall generations: the algorithm stops if the number of successive iterations without sampling a point that improves the current best solution reaches a given threshold.

- Stagnation of the cost: the algorithm stops if the absolute difference between the maximum and minimum of the objective function values over a given number of iterations (*i.e.*, its *range*) is below a given threshold.
- Stagnation of the solution: the algorithm stops if the possible change in the solution becomes extremely small, *i.e.*, the current normal distribution can only sample points that are extremely close to its mean.
- Number of function evaluations: the algorithm stops if the number of calls to the objective function reaches a given threshold.

The algorithm stops when any one of these criteria is reached.

3 Syntax

```
XSTAR = uq_c1p1cmaes(FUN, X0, SIGMA0)
XSTAR = uq_c1p1cmaes(FUN, X0, SIGMA0, LB, UB)
XSTAR = uq_c1p1cmaes(FUN, X0, SIGMA0, LB, UB, NONLCON)
XSTAR = uq_c1p1cmaes(FUN, X0, SIGMA0, LB, UB, NONLCON, OPTIONS)
[XSTAR,FSTAR] = uq_c1p1cmaes(...)
[XSTAR,FSTAR,EXITFLAG] = uq_c1p1cmaes(...)
[XSTAR,FSTAR,EXITFLAG,OUTPUT] = uq_c1p1cmaes(...)
```

`XSTAR = uq_c1p1cmaes(FUN, X0, SIGMA0)` finds a local minimizer of the function `FUN` with `X0` as starting point and `SIGMA0` as the initial global step size.

`XSTAR = uq_c1p1cmaes(FUN, X0, SIGMA0, LB, UB)` defines a set of lower and upper bounds such that `LB(i) <= XSTAR(i) <= UB(i)`. If `LB` and `UB` are finite and `X0 = []` and/or `SIGMA0 = []`, the center of the search space, i.e., `(LB(i)+UB(i))/2` and $1/6$ of the search space width, i.e., `(UB(i)-LB(i))/6` are used as `X0(i)` and `SIGMA0(i)`, respectively.

`XSTAR = uq_c1p1cmaes(FUN, X0, SIGMA0, LB, UB, NONLCON)` defines a set of non-linear inequalities constraints and subjects the minimization to the constraints. If there are no bound constraints, set `LB = []` and `UB = []`.

`XSTAR = uq_c1p1cmaes(FUN, X0, SIGMA0, LB, UB, NONLCON, OPTIONS)` minimizes with the default optimization options replaced by the values in the `OPTIONS` structure (see Table 2).

`[XSTAR,FSTAR] = uq_c1p1cmaes(...)` additionally returns the value of the objective function at the solution `XSTAR`.

`[XSTAR,FSTAR,EXITFLAG] = uq_c1p1cmaes(...)` additionally returns an exit flag that indicates the termination condition of the algorithm (see Table 4).

`[XSTAR,FSTAR,EXITFLAG,OUTPUT] = uq_c1p1cmaes(...)` additionally returns a structure with additional information about the optimization process (see Table 4).

4 Examples

4.1 Minimize constrained Rosenbrock's function

Consider a constrained minimization problem of the Rosenbrock's function:

$$\begin{aligned} f(\boldsymbol{x}) &= 100\left(x_2 - x_1^2\right)^2 + (1 - x_1)^2, \\ \text{subject to: } g(\boldsymbol{x}) &= x_1^2 + x_2^2 - 1. \end{aligned} \tag{13}$$

The minimum of the function is located at $\boldsymbol{x}^* = (0.7864, 0.6177)$ with $f(\boldsymbol{x}^*) = 0.0457$.

The following code solves the optimization problem using `uq_c1p1cmaes` by assuming default values for the starting point `X0` and initial global step size `SIGMA0` (see Table 5):

```
rng(42,'twister')     % For reproducible results
fun =  @(X) 100 .* (X(:,2) - X(:,1).^2).^2 + (1 - X(:,1)).^2;
nonlcon = @(X) X(:,1).^2 + X(:,2).^2 - 1;
lb = [-10 -10];
ub = [10 10];
xstar = uq_c1p1cmaes(fun, [], [], lb, ub, nonlcon)
```

The code produces:

```

Value of sigma below options.TolSigma
obj. value =     0.0456748

xstar =

0.7864    0.6177
```

4.2 Specify the starting point and initial global step size

By default, the starting point of `uq_c1p1cmaes` is `X0(i) = (LB(i)+UB(i))/2` and the initial global step size is `SIGMA0(i) = (UB(i)-LB(i))/6`. The following code specifies user-given values for these two parameters:

```
rng(100,'twister')     % For reproducible results
fun =  @(X) 100 .* (X(:,2) - X(:,1).^2).^2 + (1 - X(:;1)).^2;
nonlcon = @(X) X(:,1).^2 + X(:,2).^2 - 1;
lb = [-10 -10];
ub = [10 10];
x0 = [0.5 0.5];
sigma0 = [2 2];
xstar = uq_c1p1cmaes(fun, x0, sigma0, lb, ub, nonlcon);
```

The code produces:

```

Value of sigma below options.TolSigma
obj. value =     0.0456748
```

```
xstar =

0.7864    0.6177
```

5 Input

Table 1: uq_c1p1cmaes(FUN, X0, SIGMA0, LB, UB, NONLCON, OPTIONS)

●	FUN	Function handle	Objective function to be minimized.
⊕	X0	$1 \times M$ Double default: (LB+UB)/2	Starting point of the optimization algorithm.
⊕	SIGMA0	Scalar or $1 \times M$ Double default: (UB-LB)/6	Initial global step size.
⊕	LB	Scalar or $1 \times M$ Double default: -Inf	Lower bounds of the design space.
⊕	UB	Scalar or $1 \times M$ Double default: Inf	Upper bounds of the design space.
□	NONLCON	Function handle	Nonlinear inequality constraints to be satisfied.
□	OPTIONS	Table 2	Options of the algorithm.

Table 2: uq_c1p1cmaes(..., OPTIONS)

.Display	String default: 'final'	Level of output display.
	'none'	Displays no output.
	'iter'	Displays output at each iteration.
	'final'	Displays only the final output.
.MaxIter	Double default: $1000\,(M+5)^2$	Maximum number of iterations.
.MaxFunEval	Positive Integer default: Inf	Maximum number of function evaluations.
.nStallMax	Positive Integer default: $10 + 30 \cdot M$	Maximum number of stall generations.
.TolFun	Double default: 10^{-12}	Tolerance on the objective function: the algorithm stops if the *range* (*i.e.*, the absolute difference between the maximum and minimum values) of the best objective function values over .nStallMax generations is less than or equal to .TolFun.

Continued on next page

Table 2–continued from previous page

`.TolSigma`	Double default: $10^{-11} \cdot \boldsymbol{\sigma}_{\boldsymbol{x}}^{[0]}$	Tolerance on the input `x`: the algorithm stops if the current global step size `sigma` is lower than `.TolSigma`.
`.isactiveCMA`	Logical default: `true`	Specify if the covariance matrix should be actively adapted (updated), when the current path leads repeatedly to unsuccessful sample points (*i.e.*, sample points that do not improve the current best solution).
`.feasiblex0`	Logical default: `true`	If the starting point is not feasible, search for a feasible point by random sampling.
	`true`	Search for a feasible point before proceeding to the algorithm.
	`false`	Proceed to the algorithm without searching for a feasible point.
`Strategy`	Structure default: Table 3	Internal parameters of the C(1+1)-CMA-ES algorithm. They are already optimized. It is strongly advised not to modify them.

Table 3: `OPTIONS.Strategy`

`.dp`	Double default: $1 + M/2$	d_p in Eq. (7), see Arnold and Hansen (2012) for details.
`.c`	Double default: $2/(M+2)$	c in Eqs. (3) and (8), see Arnold and Hansen (2012) for details.
`.cp`	Double default: $1/12$	c_p in Eq. (6), see Arnold and Hansen (2012) for details.
`.Ptarget`	Double default: $2/11$	P_{target} in Eq. (7), see Arnold and Hansen (2012) for details.
`.ccovp`	Double default: $2/(M^2+6)$	c_{cov}^{+} in Eq. (11), see Arnold and Hansen (2012) for details.
`.cc`	Double default: $1/(M+2)$	c_c in Eq. (3), see Arnold and Hansen (2012) for details.
`.beta`	Double default: $0.1/(M+2)$	β in Eq. (4), see Arnold and Hansen (2012) for details.

Note: These seven parameters have been fine-tuned for the (1+1)-CMA-ES algorithm (Arnold and Hansen, 2012; Suttorp et al., 2009). It is not advised to modify their default values.

6 Output

Table 4: `[XSTAR,FSTAR,EXITFLAG,OUTPUT] = uq_c1p1cmaes(...)`		
`XSTAR`	$1 \times M$ Double	Optimal solution.
`FSTAR`	Double	Objective function value at the optimal solution.
`EXITFLAG`	Integer	Flag indicating the termination condition of the algorithm.
	`1`	Maximum number of generations reached.
	`2`	Maximum number of stall generations reached.
	`3`	Maximum number of function evaluations reached.
	`4`	Range of `FUN` over generations is smaller than threshold.
	`5`	Global step size `sigma` smaller than threshold.
	`<0`	No feasible solution was found.
`OUTPUT`	Structure, see Table 5	Diverse information about the optimization process.

Table 5: `[...,OUTPUT] = uq_c1p1cmaes(...)`		
`.message`	String	Exit message
`.iterations`	Integer	Total number of iterations.
`.funccount`	Integer	Total number of objective function evaluations.
`.constcount`	Integer	Total number of constraint functions evaluations.
`.History`	Structure, see Table 6	History of the optimization process over iterations.

Table 6: `OUTPUT.History`		
`.x`	Matrix Double	History of the sampled points at each iteration.

Continued on next page

Table 6–continued from previous page

`.fval`	Vector Double	History of the corresponding objective function values at each iteration.
`.gval`	Matrix Double	History of the corresponding constraint function values at each iteration.
`.sigma`	Matrix Double	History of the global step size at each iteration.
`.status`	Vector Integer	History of the state of the sampled point at each iteration.
	`-1`	Sampled point is not feasible.
	`0`	Sampled point is feasible, but it does not improve the current best solution.
	`1`	Sampled point is feasible and it improves the current best solution.

7 Notes

- In the constrained (1+1)-CMA-ES algorithm, the covariance matrix $\boldsymbol{C}$ itself is never explicitly computed nor required.
- In `uq_c1p1cmaes`, the initial values for the Cholesky decomposition $\boldsymbol{A}$ and its inverse $\boldsymbol{A}_{\text{inv}}$ are the identity matrix, which correspond to the unit-variance diagonal covariance matrix.
- The default values for `X0` and `SIGMA0` (see Table 1) are heuristics.

`uq_eval_Kernel` – Compute kernel matrix

1 Objective

Compute the kernel matrix given two input matrices $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ for a specified kernel function.

2 Algorithm

Some of the most popular kernels families are available in UQLAB. They are either classified as stationary or non-stationary kernels.

2.1 Kernel matrix

Let $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ be matrices in $\mathbb{R}^{N_1 \times M}$ and $\mathbb{R}^{N_2 \times M}$, respectively, whose row-wise elements are row vectors $\boldsymbol{x}_1^{(i)}$ and $\boldsymbol{x}_2^{(i)}$ in $\mathcal{D}_{\boldsymbol{X}} \subseteq \mathbb{R}^M$. Let k be a kernel function, such that $k : \mathcal{D}_{\boldsymbol{X}} \times \mathcal{D}_{\boldsymbol{X}} \mapsto \mathbb{R}$. Then the kernel matrix $\boldsymbol{K}$ is a $N_1 \times N_2$ matrix defined by

$$\boldsymbol{K} = \begin{bmatrix} k\left(\boldsymbol{x}_1^{(1)}, \boldsymbol{x}_2^{(1)}\right) & k\left(\boldsymbol{x}_1^{(1)}, \boldsymbol{x}_2^{(2)}\right) & \dots & k\left(\boldsymbol{x}_1^{(1)}, \boldsymbol{x}_2^{(N_2)}\right) \\ k\left(\boldsymbol{x}_1^{(2)}, \boldsymbol{x}_2^{(1)}\right) & k\left(\boldsymbol{x}_1^{(2)}, \boldsymbol{x}_2^{(2)}\right) & \dots & k\left(\boldsymbol{x}_1^{(2)}, \boldsymbol{x}_2^{(N_2)}\right) \\ \vdots & \vdots & \ddots & \vdots \\ k\left(\boldsymbol{x}_1^{(N_1)}, \boldsymbol{x}_2^{(1)}\right) & k\left(\boldsymbol{x}_1^{(N_1)}, \boldsymbol{x}_2^{(2)}\right) & \dots & k\left(\boldsymbol{x}_1^{(N_1)}, \boldsymbol{x}_2^{(N_2)}\right) \end{bmatrix} \tag{1}$$

The kernel function k is any symmetric positive function that satisfies the *Mercer's condition* (Cherkassky and Mulier, 2007). If $\boldsymbol{X}_1 = \boldsymbol{X}_2$, the resulting kernel matrix is the Gram matrix, a positive-semidefinite matrix (Shawe-Taylor and Cristianini, 2004). In the context of Gaussian process modeling, the kernel function and the resulting matrix correspond to the correlation kernel function and matrix, respectively.

2.2 Available kernel function families

More comprehensive examples of a valid kernel function can be found in Vapnik (1995) or Rasmussen and Williams (2006). The most popular ones are available in UQLAB and they are listed below:

- **Stationary kernels**: Stationary kernel functions depend only on the relative position of its two inputs. All the one-dimensional families defined below are parametrized by a kernel parameter θ, often referred to as the *characteristic length scale* parameter. The kernel functions below are defined for a pair of one-dimensional input $x, x' \in \mathbb{R}$. Their extension to multiple dimensions is given in Sections 2.3.1 and 2.3.2.
  - Linear:
  $$k_{\text{lin-s}}\left(x, x'; \theta\right) = \max\left(0, 1 - \frac{|x - x'|}{\theta}\right). \tag{2}$$
  - Exponential:
  $$k_{\text{exp}}\left(x, x'; \theta\right) = \exp\left(-\frac{|x - x'|}{\theta}\right). \tag{3}$$
  - Gaussian (or squared exponential):
  $$k_{\text{Gaussian}}\left(x, x'; \theta\right) = \exp\left(-\frac{1}{2}\left(\frac{|x - x'|}{\theta}\right)^2\right). \tag{4}$$
  - Matérn $3/2$:
  $$k_{3/2}\left(x, x'; \theta\right) = \left(1 + \frac{\sqrt{3}\,|x - x'|}{\theta}\right)\exp\left(-\frac{\sqrt{3}\,|x - x'|}{\theta}\right). \tag{5}$$
  - Matérn $5/2$:
  $$k_{5/2}\left(x, x'; \theta\right) = \left(1 + \frac{\sqrt{5}\,|x - x'|}{\theta} + \frac{5}{3}\frac{|x - x'|^2}{\theta^2}\right)\exp\left(-\frac{\sqrt{5}\,|x - x'|}{\theta}\right). \tag{6}$$
  - Custom user-defined stationary kernels: users can supply their own one-dimensional stationary kernel using a function handle (see Section 2.3.3).
- **Non-stationary kernels**: Non-stationary kernel functions depend on the absolute values of the inputs, rather on their difference. In contrast with the stationary kernels listed above, the following kernels are already defined in multi-dimension. That is, they are defined for a pair of M-dimensional inputs $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^M$. Moreover, with the exception of the non-stationary linear kernel function, non-stationary kernels are parametrized by a vector of kernel parameters $\boldsymbol{\theta}$.
  - Non-stationary linear:
  $$k_{\text{lin}}\left(\boldsymbol{x}, \boldsymbol{x}'\right) = \boldsymbol{x}^T\boldsymbol{x}'. \tag{7}$$
  - Polynomial:
  $$k_{\text{poly}}\left(\boldsymbol{x}, \boldsymbol{x}'; \boldsymbol{\theta}\right) = \left(\boldsymbol{x}^T\boldsymbol{x}' + d\right)^p, \tag{8}$$
  where $\boldsymbol{\theta} = \{d, p\}$, with $d \geq 0$ and $p \in \mathbb{N}^*$.
  - Sigmoid:
  $$k_{\text{sigmoid}}\left(\boldsymbol{x}, \boldsymbol{x}'; \boldsymbol{\theta}\right) = \tanh\left(\frac{\boldsymbol{x}^T\boldsymbol{x}'}{a} + b\right), \tag{9}$$
  where $\boldsymbol{\theta} = \{a, b\}$, with $a > 0$ and $b \leq 0$.

2.3 Properties of stationary kernels

For stationary kernels which can be cast as a function of $|x_1 - x_2|$ (so-called *radial basis form*, see Section 2.2), the following additional properties can be set by the user.

2.3.1 Kernel function types

When the input dimension M is greater than one, multi-dimensional stationary kernels can be constructed from one-dimensional stationary kernel families using the following constructions:

- *Ellipsoidal* kernel functions (Rasmussen and Williams, 2006), calculated as follows:

$$k\left(\boldsymbol{x}, \boldsymbol{x}'; \boldsymbol{\theta}\right) = k(h) \ , \ h = \sqrt{\sum_{i=1}^{M} \left(\frac{x_i - x_i'}{\theta_i}\right)^2}. \tag{10}$$

- *Separable* kernel functions (Sacks et al., 1989), calculated as follows:

$$k\left(\boldsymbol{x}, \boldsymbol{x}'; \boldsymbol{\theta}\right) = \prod_{i=1}^{M} k\left(x_i, x_i', \theta_i\right). \tag{11}$$

The function $k(\cdot)$ that appears on the right hand side of Eqs. (10) and (11) corresponds to the available one-dimensional kernel function families described in Section 2.2. The types of multi-dimensional construction above correspond to the *anisotropic* case, in which there is a unique kernel parameter for each input dimension.

2.3.2 Isotropic kernels

A kernel function is called *isotropic* when a single kernel parameter is associated with all the input dimensions. The isotropic version of the kernel function types introduced in the previous section are given by:

- Isotropic ellipsoidal kernel functions:

$$k\left(\boldsymbol{x}, \boldsymbol{x}'; \theta\right) = k\left(\frac{1}{\theta}\sqrt{\sum_{i=1}^{M} (x_i - x_i')^2}\right), \theta \in \mathbb{R}. \tag{12}$$

- Isotropic separable kernel functions:

$$k\left(\boldsymbol{x}, \boldsymbol{x}'; \theta\right) = \prod_{i=1}^{M} \left(k(x_i, x_i'; \theta\right), \theta \in \mathbb{R}. \tag{13}$$

2.3.3 Custom user-defined kernels

A custom user-defined kernel function should return a valid stationary kernel and accept three input arguments as illustrated in the function declaration below:

```
myK = my_eval_K(X1, X2, THETA)
```

The inputs X1 and X2 are matrices with arbitrary number of rows and M number of columns (M is also the input dimension). The input THETA corresponds to the length-scale parameter θ. A function handle referring to this function is then passed as an option to uq_eval_Kernel.

Note: Custom user-defined kernels are only supported for stationary kernels.

2.4 Nugget

Regardless on how a kernel matrix $\boldsymbol{K}$ is calculated, it is often inverted in practical applications. This procedure is well known to suffer from numerical instabilities, especially when the distances between the input points $\boldsymbol{x}_i$ are small. To circumvent this limitation, one can introduce a *nugget* $\boldsymbol{\nu}$, which is a set of values that are added to the main diagonal of $\boldsymbol{K}$:

$$k_{ii} = 1 + \nu_i. \tag{14}$$

3 Syntax

```
K = uq_eval_Kernel(X1, X2, THETA, OPTIONS)
```

K = uq_eval_Kernel(X1, X2, THETA, OPTIONS) computes the kernel matrix K for two inputs X1 and X2 given the kernel parameters THETA and additional options specified in the structure OPTIONS (see Table 2).

4 Examples

4.1 Create a Gaussian correlation matrix

Compute the correlation matrix for the vector $\boldsymbol{x} = \begin{pmatrix} 0 & 0.25 & 0.50 & 0.75 & 1 \end{pmatrix}^T$, with a Gaussian kernel with a correlation length of 0.25.

The following code creates the correlation matrix:

```
X = linspace(0,1,5)';
theta = 0.25;
Options.Family = 'Gaussian';
Options.Type = 'Separable';
Options.Isotropic = true;
Options.Nugget = 0;

R = uq_eval_Kernel(X, X, theta, OPTIONS)
```

The resulting matrix K is symmetric and has size 5×5:

```
R =

1.0000    0.6065    0.1353    0.0111    0.0003
0.6065    1.0000    0.6065    0.1353    0.0111
```

```
0.1353    0.6065    1.0000    0.6065    0.1353
0.0111    0.1353    0.6065    1.0000    0.6065
0.0003    0.0111    0.1353    0.6065    1.0000
```

Figure 1 compares three different correlation matrices computed by `uq_eval_Kernel` for three different correlation lengths.

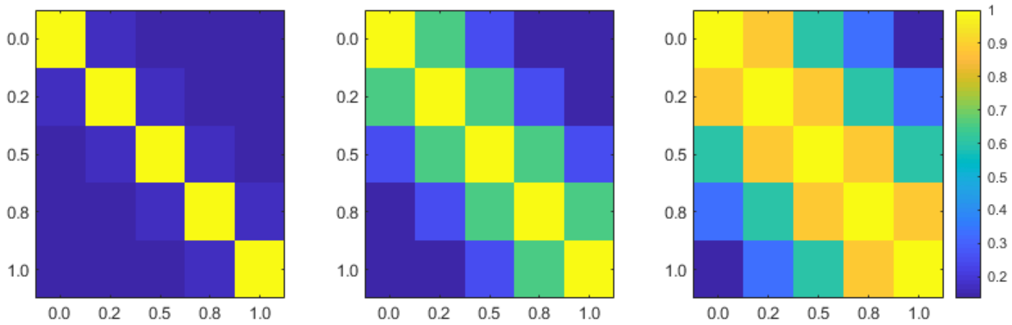


Figure 1: Plots of correlation matrices with three different correlation lengths θ: 0.1 (left), 0.25 (center), and 0.5 (right).

5 Input

Table 1: `uq_eval_Kernel(X1, X2, THETA, OPTIONS)`

●	`X1`	$N_1 \times M$ Double	First input vector.
●	`X2`	$N_2 \times M$ Double	Second input vector.
●	`THETA`	Scalar or vector of Double	Kernel function parameters, see Section 2.2. • For stationary and radial basis type kernels: `THETA` should be either a scalar or a vector of size $1 \times M$. • When `THETA` is scalar and the Kernel is anisotropic the same value is replicated in all dimensions. • For polynomial and sigmoid kernels, θ should be a vector of size 1×2.
●	`OPTIONS`	Table 2	Options of the kernel function.

Table 2: uq_eval_Kernel(..., OPTIONS)		
.Family	String or Function handle	The kernel family, see Section 2.2.
	'Linear'	Linear kernel function, see Eq. (2).
	'Exponential'	Exponential kernel function, see Eq. (3).
	'Gaussian'	Gaussian kernel function, see Eq. (4).
	'Matern-3_2'	Matérn-3/2 kernel function, see Eq. (5).
	'Matern-5_2'	Matérn-5/2 kernel function, see Eq. (6).
	'Linear-NS'	Non-stationary linear kernel function, see Eq. (7).
	'Polynomial'	Polynomial kernel function, see Eq. (8).
	'Sigmoid'	Sigmoid kernel function, see Eq. (9).
	Function handle	Custom user-defined stationary kernel function, see Eq. (8).
.Type	String	Kernel function type. Only applies to the stationary kernels, see Section 2.3.3.
	'Ellipsoidal'	Ellipsoidal kernel function.
	'Separable'	Separable kernel function.
.Isotropic	Logical	Determines whether the kernel function is isotropic or anisotropic. Only applies to the stationary kernel functions, see Sections 2.2 and 2.3.2.
.Nugget	Scalar or $1 \times M$ Double	Nugget value. Only applicable when X1 = X2 (*i.e.*, a Gram matrix). • If scalar, adds this quantity to the diagonal elements of the kernel matrix $\boldsymbol{K}$. • If vector, adds each element to the corresponding diagonal element of $\boldsymbol{K}$.

6 Output

Table 3: K = uq_eval_Kernel(...)		
K	$N_1 \times N_2$ Double	Kernel matrix. A Gram matrix is obtained when X1 = X2.

`uq_subsample_random` – Random subsampling

1 Objective

Given a sample set $\mathcal{X} = \left\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(N)}\right\}$ of size N, create a reduced set $\mathcal{X}_s \subseteq \mathcal{X}$ by randomly selecting $N_s \leq N$ sample points from $\mathcal{X}$.

2 Algorithm

Random subsampling consists in creating a subset $\mathcal{X}_S \subset \mathcal{X}$ by randomly selecting sample points from $\mathcal{X}$. That is, the subsampled experimental design $\mathcal{X}_s$ contains a reduced number of sample points $\mathcal{X}_s = \left\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(N_s)}\right\}$, $N_s \leq N$. This function may be used to create training and validation sets for cross-validation in the context of machine learning algorithms.

3 Syntax

```
XS = uq_subsample_random(X,NS)
[XS,IDX] = uq_subsample_random(...)
```

`XS = uq_subsample_random(X,NS)` returns a subset `XS` (`NS`-by-`M`) of `X` (`N`-by-`M`), based on random selection. `NS` has to be less than or equal to `N`.

`[XS,IDX] = uq_subsample_random(...)` additionally returns the indices of the selected sample points, such that `XS = X(IDX,:)`.

4 Input

Table 1: `uq_subsample_random(X,NS)`

●	`X`	$N \times M$ Double	Sample set.
●	`NS`	Integer ($N_s \leq N$)	Size of the subsample.

5 Output

Table 2: `[XS,IDX] = uq_subsample_random(...)`		
XS	$N_s \times M$ Double	Subsample.
IDX	$N_s \times M$ Integer	Indices of the randomly selected subsample points.

6 Example

This example creates a subsample the reduced Fisher's Iris data set, a 100×2 data set. The data set contains 100 observations and two variables, namely the petal width and length. The reduced Fisher's Iris data set is provided as part of the UQLab distribution. The data set can be loaded in variable X as follows:

```
load fisher_iris_reduced.mat
```

Out of the 100 sample points available in X, 10 are selected by random subsampling:

```
rng(100,'twister')  % for reproducible results
Xs = uq_subsample_random(X,10);
```

Visualize the results by plotting the two different sets (see Figure 1):

```
uq_figure()
plot(X(:,1), X(:,2), '.')
hold on
plot(Xs(:,1), Xs(:,2), 'ro')
```

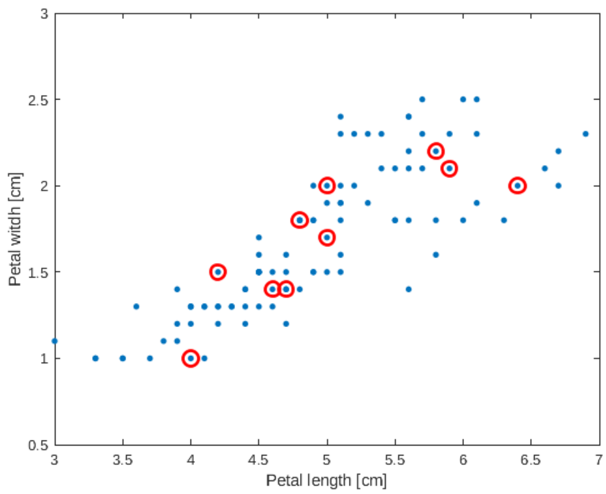


Figure 1: Random selection of 10 points from the reduced Fisher' Iris data set by `uq_subsample_random`.

uq_subsample_kmeans – k-means clustering-based subsampling

1 Objective

Given a sample set $\mathcal{X} = \left\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(N)}\right\}$ of size N, create a reduced set $\mathcal{X}_s \subseteq \mathcal{X}$ by creating $N_s \leq N$ clusters and, for each cluster, selecting the nearest point to the cluster centroid.

2 Algorithm

The k-means subsampling algorithm works as follows:

1. Identify N_s clusters centroids in $\mathcal{X}$ using the k-means clustering algorithm with $k = N_s$ (Lloyd, 1982).

2. For each cluster centroid, the nearest sample point determined by the k-nearest-neighbor-search algorithm with $k = 1$ is included in $\mathcal{X}_s$ (Friedman et al., 1977).

3 Syntax

```
XS = uq_subsample_kmeans(X,NS)
XS = uq_subsample_kmeans(X, NS, NAME, VALUE)
[XS,IDX] = uq_subsample_kmeans(...)
```

`XS = uq_subsample_kmeans(X,NS)` returns a subset `XS` (`NS`-by-`M`) of `X` (`N`-by-`M`).

`XS = uq_subsample_kmeans(X, NS, NAME, VALUE)` allows for fine-tuning various parameters of the subsampling algorithm by specifying `NAME` and `VALUE` pairs of options. The available options are summarized in Table 2.

`[XS,IDX] = uq_subsample_kmeans(...)` additionally returns the indices of the selected sample points, such that `XS = X(IDX,:)`.

4 Input

Table 1: uq_subsample_kmeans(X, NS, NAME, VALUE)

●	X	$N \times M$ Double	Sample set.
●	NS	Integer ($N_s \leq N$)	Size of the subsample.
□	NAME, VALUE	name-value pair, see Table 2	Additional options.

Table 2: uq_subsample_kmeans(..., NAME, VALUE)

'Distance_kmeans'	String default: 'sqeuclidean'	Distance measure used in the k-means clustering. List of the available options can be found in the documentation of the built-in MATLAB function kmeans (see the option 'Distance').
'Distance_nn'	String default: 'euclidean'	Distance measure used in the nearest-neighbor search for determining the sample points closest to the k-means centroids. List of available options can be found in the documentation of the built-in MATLAB function knnsearch (see option 'Distance').

5 Output

Table 3: [XS,IDX] = uq_subsample_kmeans(...)

XS	$N_s \times M$ Double	Subsample.
IDX	$N_s \times M$ Integer	Indices of the randomly selected subsample points.

6 Examples

This example subsamples the reduced Fisher's Iris data set, a 100×2 data set. The data set contains 100 observations and two variables, namely the petal width and length. The reduced Fisher's Iris data set is provided as part of the UQLAB distribution. The data set can be loaded in variable X as follows:

```
load fisher_iris_reduced.mat
```

Out of the 100 sample points available in X, 10 are selected by k-means clustering using the default options:

```
rng(100,'twister')  % for reproducible results
```

```
Xs = uq_subsample_random_kmeans(X,10);
```

Visualize the results by plotting the two different sets (see Figure 1):

```
uq_figure()
plot(X(:,1), X(:,2), '.')
hold on
plot(Xs(:,1), Xs(:,2), 'ro')
```

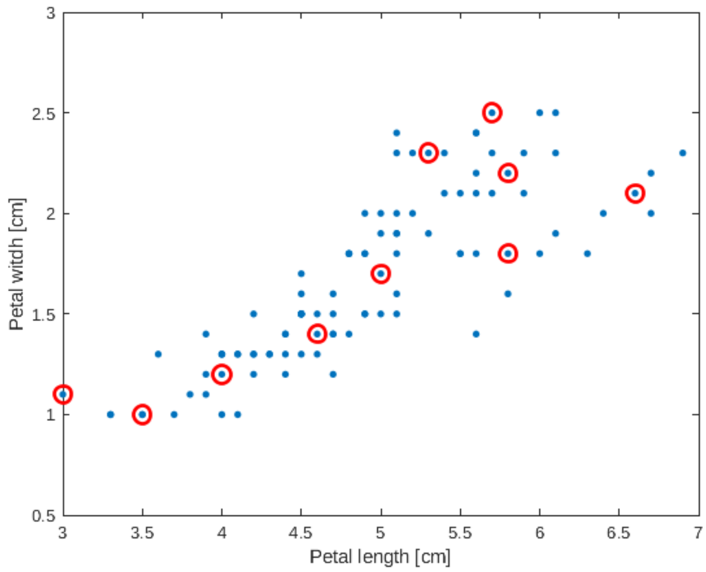


Figure 1: Selection of 10 points from the reduced Fisher's Iris data set by uq_subsample_kmeans.

7 Notes

- Strictly speaking, this method is more appropriately referred to as the *k-medoid sampling*.
- This function utilizes MATLAB functions `kmeans` and `knnsearch` for the k-means clustering and the nearest-neighbor search, respectively. Both functions are part of the Statistics and Machine Learning Toolbox.

uq_figure – Create a figure

1 Objective

Create a figure window following the default formatting and screen placement of UQLab.

2 Description

uq_figure wraps the MATLAB figure command and sets its formatting style following UQLab defaults. By default, it also places a newly created figure in the center of the screen. The input arguments to uq_figure (and most of their default values) as well as the resulting figure behavior are consistent with the figure command. A list of default properties specifically set by uq_figure is available in Table 1.

Table 1: uq_figure defaults

'Color'	'white'
'Position'	centered (depends on the actual screen size) with a width and height of 798 and 600 pixels, respectively.
'Renderer'	'opengl'
'Name'	'uq_figure_(figure number)' (the figure number uses an internal counter)
'Filename'	'uq_figure_(figure number).fig' (the figure number uses an internal counter)

3 Syntax

```
uq_figure
uq_figure(...)
uq_figure(..., NAME, VALUE)
F = uq_figure(...)
```

uq_figure creates a new formatted figure window.

uq_figure(...) wraps the MATLAB figure command, supporting the same input arguments.

`uq_figure(..., Name, Value)` also sets the properties of the figure according to the specified NAME/VALUE pairs.

`F = uq_figure(...)` returns the `Figure` object. Use `F` to access or modify the properties of the figure after it is created. In MATLAB R2014a or older, `F` is the handle to the `Figure` object.

4 Examples

4.1 Create a figure with default formatting

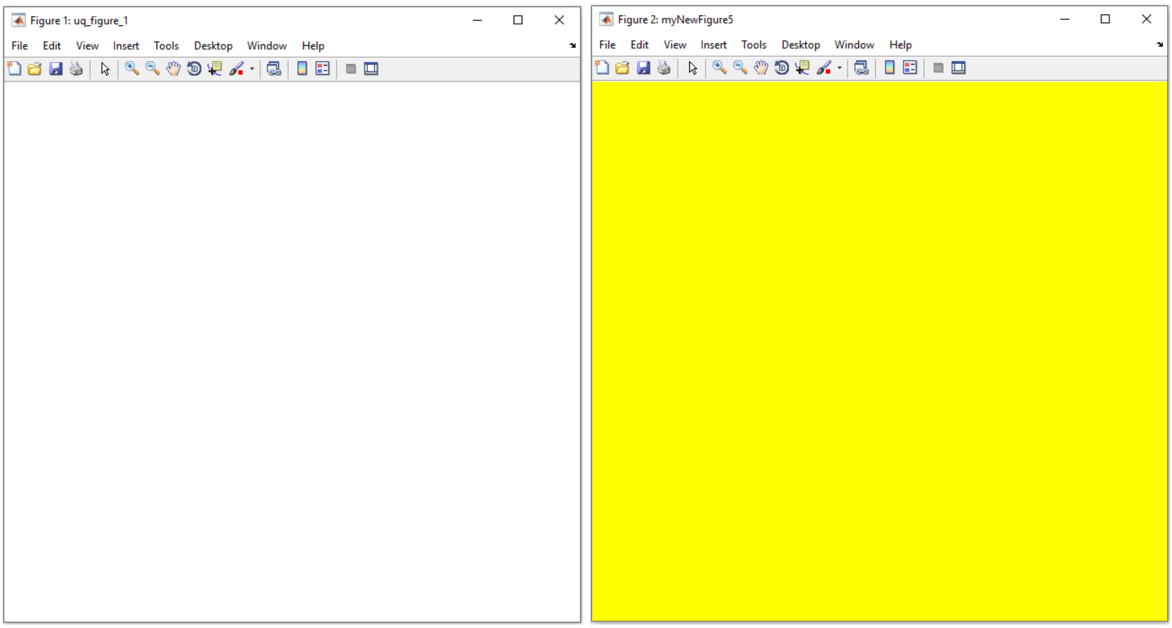
A new figure window following the default formatting of UQLAB can be opened using the `uq_figure` command (see Figure 1(a) for the resulting figure):

```
uq_figure
```

4.2 Formatted figure

To modify the appearance of a figure window, the standard MATLAB name-value pairs can be used. For example (see Figure 1(b) for the resulting figure):

```
uq_figure('Color', 'y', 'Name', 'myNewFigure5')
```



(a) Default formatting

(b) Custom formatting

Figure 1: Figures with default and custom formatting.

5 Input

The input arguments for `uq_figure` are identical to those of the standard `figure` command in MATLAB. Refer to its documentation for details.

6 Output

Table 2: `F = uq_figure(...)`

`F`	`Figure` object	The object can be used to query or modify the properties of the figure. In MATLAB R2014a or older, `F` is a handle (*i.e.*, unique identifiers) to the `Figure` object.

7 Notes

- In MATLAB R2014a or older, the function returns a handle to the `Figure` object. By default, this figure handle has the same value as the figure number. In MATLAB R2014b or newer, the function returns directly the `Figure` object.

uq_formatDefaultAxes – Default formatting of an *Axes* object

1 Objective

Format an Axes object following UQLAB defaults.

2 Description

uq_formatDefaultAxes formats a given Axes object according to the UQLAB default formatting style given in Table 1.

Table 1: uq_formatDefaultAxes defaults (Axes object)

'Box'	'on'
'FontSize'	Computed based on the size of the current figure
'Layer'	'top'
'LineWidth'	1.5
'TickLabelInterpreter'	'LaTeX'
'XGrid'	'on'
'YGrid'	'on'
'ZGrid'	'on'

Besides setting the properties of the Axes object, uq_formatDefaultAxes also sets the default properties of several children Text objects: 'XLabel', 'YLabel', 'ZLabel', and 'Title'. The default properties for these objects are listed in Table 2.

Table 2: uq_formatDefaultAxes defaults (Text objects)

'FontSize'	Computed based on the size of the current figure. This value is identical to the one for the 'FontSize' property of the Axes object.
'Interpreter'	'LaTeX'

3 Syntax

```
uq_formatDefaultAxes(AX)
uq_formatDefaultAxes(..., NAME, VALUE)
```

`uq_formatDefaultAxes(ax)` formats the `Axes` object `AX` according to the UQLAB default formatting style.

`uq_formatDefaultAxes(..., NAME, VALUE)` uses `NAME/VALUE` pairs to override the defaults.

4 Examples

4.1 Create an Axes with default formatting

In the example below, a figure is created with an `Axes` which is then formatted by `uq_formatDefaultAxes`. The result is shown in Figure 1(a).

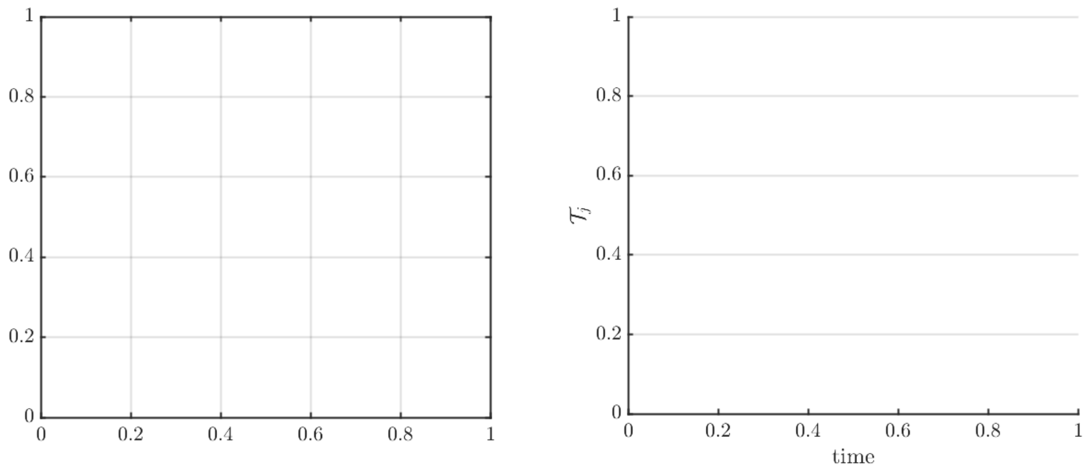
```
uq_figure
uq_formatDefaultAxes(gca)
```

4.2 Create an Axes with custom formatting

To modify the `Axes` or override the defaults, a set of name-value pairs can be used. For example:

```
uq_figure
uq_formatDefaultAxes(gca, 'XGrid', 'off', 'Box', 'off')
xlabel('time')
ylabel('$\mathcal{T}_j$')
```

will result in Figure 1(b).



(a) Default formatting (b) Custom formatting

Figure 1: Axes with default and custom formatting.

5 Input

Table 3: uq_formatDefaultAxes(AX)			
●	AX	Axes object	Target Axes.
☐	NAME, VALUE	name-value pair	Additional options as name-value pairs. They can be used to override the defaults set according to Table 1.

6 Output

uq_formatDefaultAxes does not return any output.

7 Notes

- The Axes property 'TickLabelInterepreter' is not available in MATLAB R2014a or older.

uq_plot – Create a formatted 2D line plot

1 Objective

Create a 2-dimensional line plot following the default formatting of UQLab.

2 Description

uq_plot wraps the Matlab plot command and sets the default formatting style of UQLab on the resulting Figure and Axes objects. The input arguments of uq_plot conform to the standard Matlab plot function. The properties specifically set by uq_plot are given in Table 1.

Table 1: uq_plot defaults

'LineWidth'	2.0
'MarkerFaceColor'	Same as the line 'Color' property

3 Syntax

```
uq_plot(X,Y)
uq_plot(...)
uq_plot(..., NAME, VALUE)
uq_plot(AX,...)
H = uq_plot(...)
```

uq_plot(X,Y) plots vector Y versus vector X following UQLab default formatting.

uq_plot(...) wraps the Matlab plot function, supporting the same input arguments.

uq_plot(..., NAME, VALUE) modifies the properties of the plot according to the specified NAME/VALUE pairs. The complete list of NAME/VALUE pairs can be found in the documentation of the Matlab plot function.

uq_plot(AX,...) creates the plot into the Axes object AX instead of the current axes.

H = uq_plot(...) returns a column vector of Line objects. Use the elements in H to access modify the properties of specific plot elements after they are created. In MATLAB R2014a or older, the function returns one or more handles to `lineseries` objects.

4 Examples

4.1 Create a line plot with default formatting

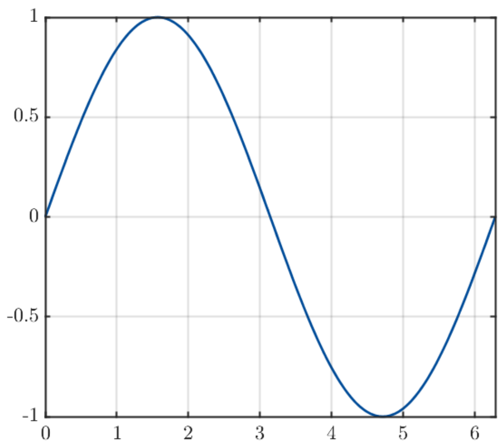
This example creates the 2-dimensional line plot following the default formatting of UQLAB. The resulting plot is shown in Figure 1(a).

```
x = 0:pi/100:2*pi;
y = sin(x);
uq_plot(x,y)
```

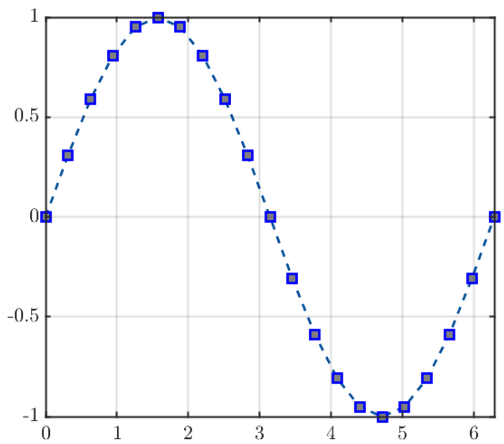
4.2 Create a line plot with custom formatting

This examples creates the 2-dimensional line plot with custom formatting shown Figure 1(b).

```
x = 0:pi/10:2*pi;
y = sin(x);
uq_plot(...
  x, y, '--s',...
  'LineWidth', 2,...
  'MarkerSize', 10,...
  'MarkerEdgeColor', 'b',...
  'MarkerFaceColor', [0.5 0.5 0.5])
```



(a) Default formatting



(b) Custom formatting

Figure 1: A 2-dimensional line plot with default and custom formatting.

Note that all the valid input arguments are consistent with the `plot` function in MATLAB.

5 Input

The input arguments for uq_plot are identical to the ones for the MATLAB's plot function. Refer to its documentation for details.

6 Output

Table 2: H = uq_plot(...)

H	Scalar or vector of Line Objects	Use the object to access and modify the properties of a specific line. In MATLAB R2014a or older, they are unique identifiers (*i.e.*, handles) to lineseries objects.

7 Note

- Prior to plotting, uq_plot searches for the current active figure. If none is found, a new figure window is created using uq_figure. It then searches for an Axes object. If none is found, a new axes is created and formatted according to the UQLAB defaults by uq_formatDefaultAxes.
- In MATLAB R2014a or older, the function returns one or more handles to lineseries object. In MATLAB R2014b or newer, the function returns one or more Line objects.
- uq_plot function wraps the MATLAB plot function and it accepts all the input arguments of the MATLAB function. The default of uq_plot can be overridden by passing other values to the default properties (Table 1) during the function call.

uq_bar – Create a bar plot

1 Objective

Create a bar graph following the default formatting of UQLAB.

2 Description

uq_bar wraps the MATLAB bar function and sets the default formatting style of UQLAB on the resulting Figure and Axes objects. The input arguments to uq_bar follow those of the default bar function in MATLAB. The properties specifically set by uq_bar can be found in Table 1.

Table 1: uq_bar defaults	
'EdgeColor'	'none'

3 Syntax

```
uq_bar(Y)
uq_bar(X,Y)
uq_bar(..., NAME, VALUE)
uq_bar(AX,...)
uq_bar(...)
H = uq_bar(...)
```

uq_bar(Y) creates a set of bar graphs, one for each element in Y, following the default formatting of UQLAB.

uq_bar(X,Y) creates a set of bar graphs, one for column of Y at the locations specified in X.

uq_bar(...) wraps the MATLAB bar function, supporting the same input arguments.

uq_bar(..., NAME, VALUE) modifies the properties of the plot according to the specified NAME/VALUE pairs. The complete list of the available NAME/VALUE pairs can be found in the documentation of the MATLAB bar function.

uq_bar(AX,...) creates the plot into the axes AX instead of the current axes.

H = uq_bar(...) returns one or more Bar objects. Use the elements in H to access and modify the properties of a specific `Bar` object after it has been created. In MATLAB R2014a or older, the function returns one or more handles to `barseries` objects.

4 Examples

4.1 Create a bar plot with default formatting

A simple bar plot with the UQLAB default formatting style is created in the example below. The resulting plot is shown in Figure 1(a).

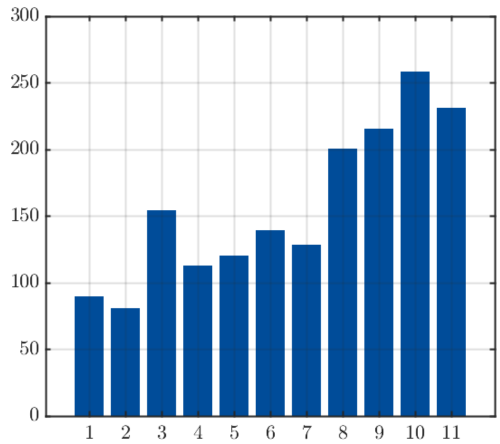
```
y = [90 81 155 113.5 121 140 129 201 216 259 231.5];
uq_bar(y)
```

4.2 Create a bar plot with custom formatting

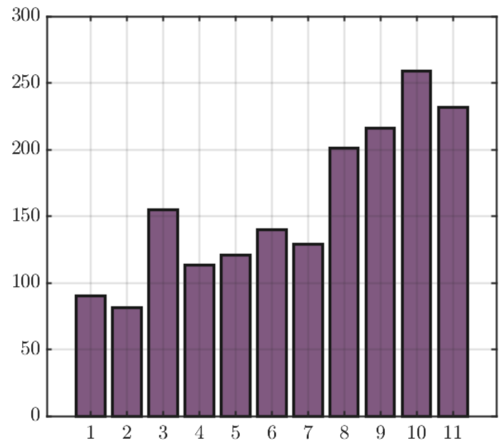
A bar plot can be further customized as illustrated in the example below. The resulting plot is shown in Figure 1(b).

```
y = [90 81 155 113.5 121 140 129 201 216 259 231.5];
uq_bar(y,...
  'FaceColor', [0.5 0.35 0.5],...
  'EdgeColor',[0.1 0.1 0.1],...
  'LineWidth', 2.5)
```

Note that all the valid input arguments are consistent with MATLAB `bar` function.



(a) Default formatting



(b) Custom formatting

Figure 1: A bar plot with default and custom formatting style.

4.3 Create a bar plot with different bar locations

The location of the bars can be specified as illustrated with the example below. The resulting plot is shown in Figure 2.

```
x = 2014:1:2019;
y = [75 91 105 123.5 131 281.5];
uq_bar(x,y)
```

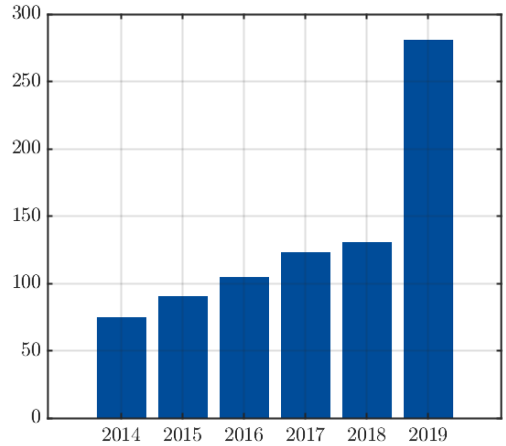


Figure 2: A bar plot with custom locations of the bars.

5 Input

The input arguments for `uq_bar` are identical to the ones for the MATLAB's `bar` function. Refer to its documentation for details.

6 Output

Table 2: `H = uq_bar(...)`

`H`	Scalar or vector of `Bar` Objects	Use the object to access and modify the properties of a specific `bar` plot. In MATLAB R2014a or older, they are handles (*i.e.*, unique identifiers) to `barseries` objects.

7 Notes

- In MATLAB R2014a or older, the function returns one or more handles to `barseries` objects. In MATLAB R2014b or newer, the function returns one or more `Bar` objects.

`uq_histogram` – Create a histogram

1 Objective

Create a histogram using the default formatting of UQLab.

2 Description

Given a set of univariate data points, `uq_histogram` creates a histogram, a type of bar plot that groups the data into bins. By default, the resulting histogram is *normalized*, that is, the area under the histogram is summed up to 1.0. `uq_histogram` uses an automatic binning algorithm that returns bins with uniform width. The width of the bins are computed by Scott's rule (Scott, 2010)

$$w = 3.49\, \hat{\sigma}_y\, n^{-1/3} \tag{1}$$

where w is the width of the bins; n is the number of data points; and $\hat{\sigma}_y$ is the empirical standard deviation of the data points.

3 Syntax

```
uq_histogram(Y)
uq_histogram(X,Y)
uq_histogram(..., 'Normalized', false)
uq_histogram(..., NAME, VALUE)
uq_histogram(AX,...)
H = uq_histogram(...)
[H,N] = uq_histogram(...)
[H,N,X] = uq_histogram(...)
```

`uq_histogram(Y)` plots a histogram of a column- or row-vector `Y` with normalized values (*i.e.*, total area under the histogram is 1.0). `uq_histogram` uses an automatic binning algorithm that returns bins with a uniform width.

`uq_histogram(X,Y)` plots a histogram of `Y` among bins with the centers specified by vector `X`.

`uq_histogram(..., 'Normalized', false)` plots an unnormalized (*i.e.*, raw counts) histogram. By default, the value of the named argument `'Normalized'` is `true`.

`uq_histogram(..., NAME, VALUE)` modifies the properties of the plot according to the specified NAME/VALUE pairs. The complete list of the available NAME/VALUE can be found in the documentation of the MATLAB bar function.

uq_histogram(AX,...) creates the plot into the Axes object AX instead of the current axes.

H = uq_histogram(...) returns the underlying Bar object used to create the plots. Use H to access and modify the properties of the underlying Bar object after it has been created. In MATLAB R2014a or older, the function returns the handle to barseries object.

[H,N] = uq_histogram(...) additionally returns the number of elements per bins (containers). The values might be normalized depending on the option set by the named argument 'Normalized'.

[H,N,X] = uq_histogram(...) additionally returns the center position of the bins along the x-axis.

4 Examples

4.1 Create a default histogram

A simple histogram plot following the default formatting of UQLAB is created in the example below. The resulting plot is shown in Figure 1(a).

```
y = randn(1000,1);
uq_histogram(y)
```

4.2 Create a customized histogram

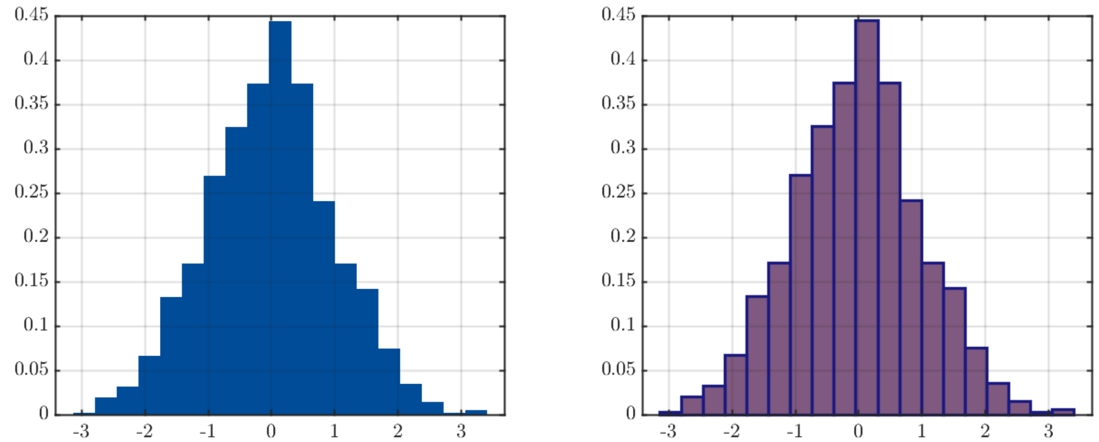
A histogram created by uq_histogram can be further customized as illustrated in the example below. The resulting plot is shown in Figure 1(b).

```
y = randn(1000,1);
uq_histogram(...
    y,...
    'FaceColor', [0.5 0.35 0.5],...
    'EdgeColor',[0.1 0.1 0.5],...
    'LineWidth', 2.5)
```

4.3 Create multiple histograms

Multiple histograms can be created in the same figure as illustrated in the example below in the case of two histograms. The resulting plot is shown in Figure 2.

```
y1 = randn(1000,1);
y2 = 2 + 2*randn(1000,1);
uq_histogram(y1)
hold on
uq_histogram(y2, 'FaceAlpha', 0.8)
```



(a) Default formatting (b) Custom formatting

Figure 1: A histogram with default and custom formatting style.

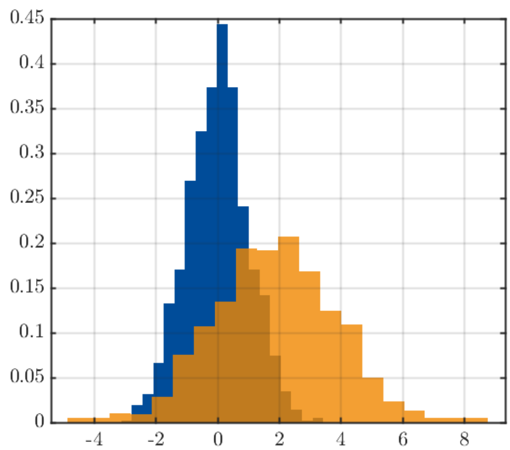


Figure 2: Multiple histograms.

4.4 Create a raw count histogram

By default, the histogram created by `uq_histogram` is *normalized* such that the area under the histogram is equal to 1.0. To create a histogram with raw counts, use the example below with the resulting plot shown in Figure 3.

```
y = randn(1000,1);
uq_histogram(y, 'Normalized', false)
```

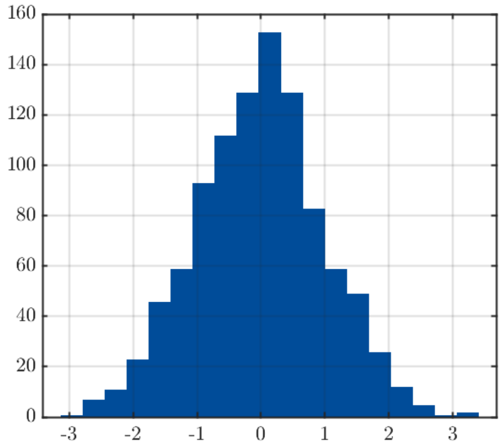


Figure 3: A histogram with raw counts.

5 Input

Table 1: `uq_histogram(...)`

●	Y	$1 \times N$ or $N \times 1$ Double	Vector of input data
□	X	$1 \times K$ or $K \times 1$ Double	Vector of bin centers along the $x-$axis.
□	AX	`Axes` object	`Axes` object in which the histogram will be created. If not specified, `uq_histogram` uses the current axes for the histogram.
□	NAME, VALUE	name-value pair (See Table 2)	Additional options as name-value pairs.

Table 2: `uq_histogram(..., NAME, VALUE)`

'Normalized'	Logical default: `true`	Histogram normalization. If normalized, the total area of the histogram equals 1.0.
'ColorRange'	1×2 Double	A pair that defines the range used to color the histogram bars. The bars in the histogram are colored according to their height by linearly interpolating between the specified color pair. This option is only supported in MATLAB R2017b or newer; In older MATLAB, the option is ignored.

Note: A histogram constructed by `uq_histogram` consists of bar elements created by `uq_bar`. Therefore, all the name-value pairs valid for `uq_bar` are also valid for `uq_histogram`.

6 Output

Table 3: `[H,N,X] = uq_histogram(...)`

`H`	`Bar` Object	Use the object to access and modify the properties of the histogram. In MATLAB R2014a or older, it is the handle (*i.e.*, unique identifier) to a `barseries` object.
`N`	Vector of Double	Number of elements (data points) per bins. The values might be normalized depending on the value of the `'Normalized'` option.
`X`	Vector of Double	Center positions of the bins along the x-axis.

7 Notes

- In MATLAB R2014a or older, the function returns the handle to a `barseries` object. In MATLAB R2014b or newer, the function returns a `Bar` object.

uq_violinplot – Create violin plots

1 Objective

Create violin plots.

2 Description

A violin plot is used to visualize the distribution of a data set. The plot is similar in principle to a Tuckey box-plot, but instead only showing interquantile ranges it shows kernel-density estimates of the data density. The plot is especially useful to visualize multimodal and highly skewed distributions.

3 Syntax

```
uq_violinplot(Y)
uq_violinplot(X,Y)
uq_violinplot(..., NAME, VALUE)
uq_violinplot(AX,...)
H = uq_violinplot(...)
```

`uq_violinplot(Y)` generates `M` a violin plot based on the `N`-by-`M` data matrix `Y`.

`uq_violinplot(X,Y)` generates violin plots centered on specified points in `X`. The number of elements in `X` must be consistent with the number of columns in `Y`.

`uq_violinplot(..., NAME, VALUE)` modifies the properties of the plot according to the specified NAME/VALUE pairs. The complete list of `NAME/VALUE` pairs can be found in the documentation of the MATLAB `patch` function.

`uq_violinplot(AX,...)` creates the plot into the axes `AX` instead of the current axes.

`H = uq_violinplot(...)` returns one or more `Patch` objects. Use the elements in `H` to access and modify the properties of a specific violin plot. In MATLAB R2014a or older, the function returns one or more handles to `patch` objects.

4 Examples

4.1 Create a violin plot with default formatting

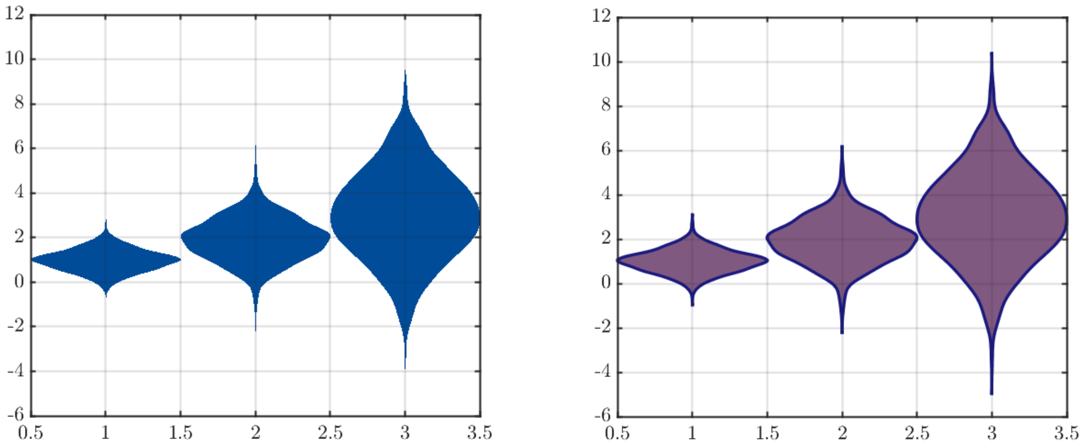
A series of violin plots following the default formatting style of UQLAB is created in the example below. The resulting plot is shown in Figure 1(a).

```
y = [1 2 3] + [0.5 1.0 2] .* randn(1000,3);
uq_violinplot(y)
```

4.2 Create a violin plot with custom formatting

A series of violin plots can be further customized as illustrated in the example below. The resulting plot is shown in Figure 1(b).

```
y = [1 2 3] + [0.5 1.0 2] .* randn(1000,3);
uq_violinplot(...
  y,...
  'FaceColor', [0.5 0.35 0.5],...
  'EdgeColor',[0.1 0.1 0.5],...
  'LineWidth', 2.5)
```



(a) Default formatting

(b) Custom formatting

Figure 1: Violin plots with default and custom formatting style.

4.3 Create multiple violin plots

Multiple series of violin plots can be created as illustrated in the example below. The resulting plot is shown in Figure 2.

```
rng(100,'twister')  % for reproducibility
y1 = [1 2 3] + betarnd(0.5, 0.5, [1000 3]);         % Bimodal beta
y2 = [1 2 3] + [0.15 0.25 0.30] .* randn(1000,3);  % Gaussian
cmap = uq_colorOrder(2);
uq_violinplot(y1)
hold on
```

```
uq_violinplot(y2, 'FaceColor', cmap(2,:), 'FaceAlpha', 0.8)
hold off
```

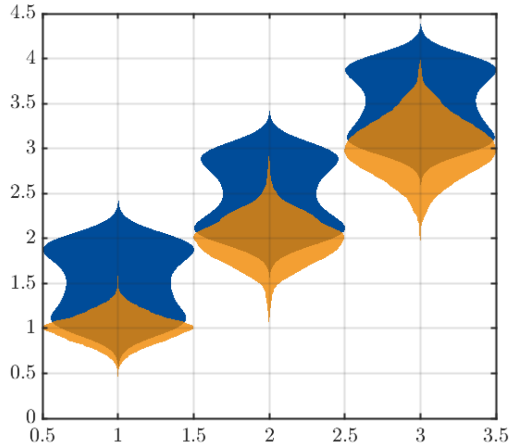


Figure 2: A bar plot with custom locations of the bars.

5 Input

Table 1: `uq_violinplot(...)`

●	`Y`	$N \times M$ Double	Data matrix, M violin plots are created for each column of `Y`.
□	`X`	$1 \times M$ or $M \times 1$ Double	The center positions of each violin plot along the x-axis.
□	`AX`	`Axes` object	`Axes` object in which the violin plots will be created. If not specified, the `uq_violinplot` uses the current axes for the plot.
□	`NAME, VALUE`	name-value pair (See Table 2)	Additional options as name-value pairs.

Table 2: `uq_violinplot(..., NAME, VALUE)`

`'FaceColor'`	1×3 Double default: `uq_colorOrder(1)`	Vector specifying the filling color of the violin plot. The color is specified in the RGB scale.
`'EdgeColor'`	1×3 Double or String default: `'none'`	Vector specifying the edge color of the violin plots. The color is specified in the RGB scale.

Note: A violin plot consists of patch elements created by MATLAB `patch` function. Therefore, all the name-value pairs valid for `patch` are also valid for `uq_violinplot` .

6 Output

Table 3: `H = uq_violinplot(...)`

`H`	Scalar or vector of `Patch` objects	Use the object to access and modify the properties of a specific violin plot. In MATLAB R2014a or older, they are handles (*i.e.*, unique identifiers) to `patch` objects.

7 Notes

- In MATLAB R2014a or older, the function returns one or more handles to `patch` objects. In MATLAB R2014b or newer, the function returns one or more `Patch` objects.
- The `'FaceColor'` property of violin plots is not cycled through using the `hold on` statement. Plotting multiple violin plots on the same figure, therefore, require manually setting their color.

uq_scatterDensity – Create a scatter plot matrix

1 Objective

Create a scatter plot matrix of multivariate data.

2 Description

Given an $N \times M$ multivariate data set $\boldsymbol{X}$, uq_scatterDensity creates a scatter plot matrix, a collection of $M \times M$ plots organized into a matrix. The diagonal elements of the scatter plot matrix visualize the histograms of each column of the input data. The (lower) off-diagonal elements of the plot visualizes the pairwise scatter plots of the corresponding columns of X. The (upper) off-diagonal elements of a scatter plot matrix created by uq_scatterDensity are empty.

3 Syntax

```
uq_scatterDensity(X)
uq_scatterDensity(..., hist_NAME, VALUE, scatter_NAME, VALUE)
uq_scatterDensity(..., NAME, VALUE)
uq_scatterDensity(AX,...)
H = uq_scatterDensity(...)
[H,AXS] = uq_scatterDensity(...)
[H,AXS,BIGAX] = uq_scatterDensity(...)
```

`uq_scatterDensity(X)` creates a scatter plot matrix from each pairs of columns in X. If X is N-by-M matrix, uq_scatterDensity produces a lower triangular matrix of M-by-M plots, in which histograms of each column of X are plotted in the main diagonal and scatter plots of each pair of columns of X are plotted as the off-diagonal elements.

`uq_scatterDensity(...,hist_NAME, VALUE, scatter_NAME, VALUE)` modifies the scatter plot matrix using additional NAME/VALUE pair arguments. The following convention applies when providing pairs to the corresponding elements of the matrix of plots:

- If NAME has the prefix 'hist_', then VALUE is passed to the function uq_histogram to create the histograms in the main diagonal;

- If NAME has the prefix 'scatter_', then VALUE is passed instead to the uq_plot function that creates the off-diagonal scatter plots.

uq_scatterDensity(..., NAME, VALUE) modifies the overall plot using NAME/VALUE pair arguments of which the NAMEs have neither the prefix 'hist_' nor 'scatter_'.

uq_scatterDensity(AX,...) creates the plot into the Axes object AX instead of the current axes.

H = uq_scatterDensity(...) returns the graphics objects of the plot. The main diagonal elements (histograms) are represented by Bar objects and the off-diagonal elements are represented by Line objects. Use the elements in H to access and modify the properties of specific plots. In MATLAB R2014a or older, the function returns handles to barseries objects (instead of Bar) and lineseries objects (instead of Line).

[H,AXS] = uq_scatterDensity(...) additionally returns the Axes objects (or handles) of all the plots inside the scatter plot matrix.

[H,AXS,BIGAX] = uq_scatterDensity(...) additionally returns the parent axes of the scatter plot matrix.

4 Examples

4.1 Create a scatter plot matrix with default formatting

A scatter plot matrix following the default formatting style of UQLAB is created in the example below. The resulting plot is shown in Figure 1.

```
rng(100,'twister')  % for reproducibility
x = mvnrnd([1 -1 3], [0.9 0.4 0.0; 0.4 0.3 0.0; 0.0 0.0 1.0], 1000);
uq_scatterDensity(x)
```

4.2 Create a scatter plot matrix with custom formatting

A scatter plot matrix can be further customized as illustrated in the example below. The resulting plot is shown in Figure 2.

```
rng(100,'twister')  % for reproducibility
x = mvnrnd([1 -1 3], [0.9 0.4 0.0; 0.4 0.3 0.0; 0.0 0.0 1.0], 1000);
uq_scatterDensity(...
  x,...
  'hist_FaceColor', [0.5 0.35 0.5],...
  'hist_EdgeColor', 'k',...
  'scatter_MarkerEdgeColor', [0.9020 0.3333 0.0510],...
  'labels', {'$X_1$', '$X_2$', '$X_3$'})
```

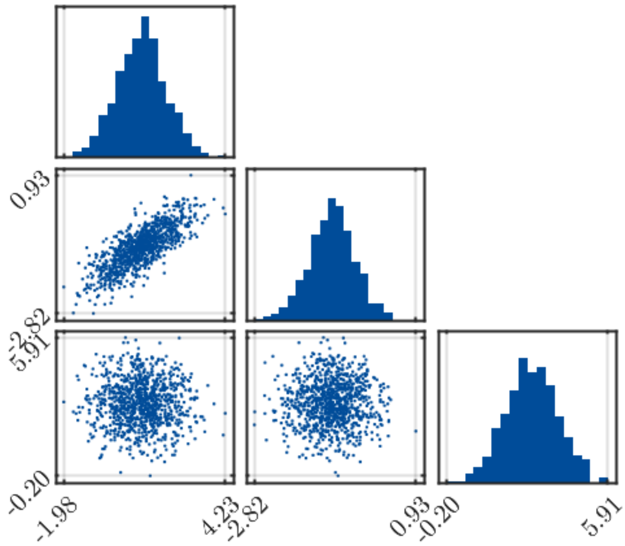


Figure 1: A scatter plot matrix with default formatting style.

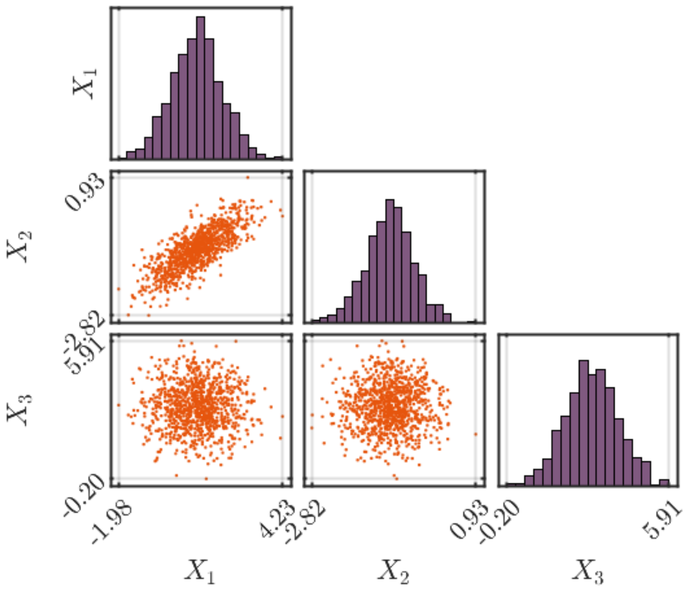


Figure 2: A scatter plot matrix with custom formatting style.

5 Input

Table 1: uq_scatterDensity(...)

●	X	$N \times M$ Double	Input data matrix
□	AX	Axes object	Axes object in which the violin plots will be created. If not specified, uq_scatterDensity uses the current axes for the plot.
□	hist_NAME, hist_VALUE	name-value pair	Additional options as name-value pairs for the histograms in the main diagonal (see uq_histogram).
□	scatter_NAME, scatter_VALUE	name-value pair	Additional options as name-value pairs for the scatter plots in the off-diagonal (see uq_plot).
□	NAME, VALUE	name-value pair (See Table 2)	Additional options as name-value pairs

Table 2: uq_scatterDensity(..., NAME, VALUE)

'points'	$N_1 \times M$ Double	A set of N_1 points that are plotted together with X in the scatter plots
'labels'	$1 \times M$ Cell Array	Labels for the subplots
'color'	1×3 Double	Vector specifying the color of the plotted elements. The color is specified in the RGB scale.
'title'	String	Title string to add to the scatter plot matrix

6 Output

Table 3: [H,AXS,BIGAX] = uq_scatterDensity(...)

H	$M \times M$ objects array	Graphics objects of the plots: Bar and Line objects for the histograms and scatter plots, respectively. In MATLAB R2014a or older, they are handles (*i.e.*, unique identifiers) to barseries and lineseries objects.
AXS	$M \times M$ Axes objects array	Axes objects (or handles) of all plots inside the scatter plot matrix
BIGAX	Axes Object	Parent axes of the scatter plot matrix

`uq_traceplot` – Create trace plots

1 Objective

Create trace plots of series of points with kernel density-based marginal estimates .

2 Description

Given a matrix of $N \times M$ random realizations $\boldsymbol{X}$, trace plots show the sampled values over consecutive realizations for each column of $\boldsymbol{X}$. For each trace plot, a kernel density estimate plot of all realizations is also plotted side by side. Trace plots are typically used in the visual assessment of Markov Chain Monte Carlo simulation to inspect the behavior and convergence of the chain (see also UQLAB User Manual – Bayesian inference for model calibration and inverse problems).

3 Syntax

```
uq_traceplot(X)
uq_traceplot(..., NAME, VALUE)
uq_traceplot(AXES,...)
PLOTAX = uq_traceplot(...)
```

`uq_traceplot(X)` plots the trace plots of an `N`-by-`M`-by-`C` array `X`. By default, `M` figures are created.

`uq_traceplot(..., NAME, VALUE)` modifies the properties of the trace plot according to the specified NAME/VALUE pair.

`uq_traceplot(AXES,...)` plots into a cell array of `Axes` objects `AXES`.

`PLOTAX = uq_traceplot(....)` returns the `Axes` objects of the generated plot in an `M`-by-2 cell array. In MATLAB R2014a or older, the function returns a cell array with handles to the `Axes` objects.

4 Examples

4.1 Create trace plots

Two trace plots based on 500 realizations of a bivariate normal distribution are created following the default formatting of UQLab in the example below.

```
rng(100,'twister')  % for reproducibility
x = mvnrnd([1 -1], [0.9 0.4; 0.4 0.3], 500);
uq_traceplot(x)
```

The resulting plot is shown in Figure 1.

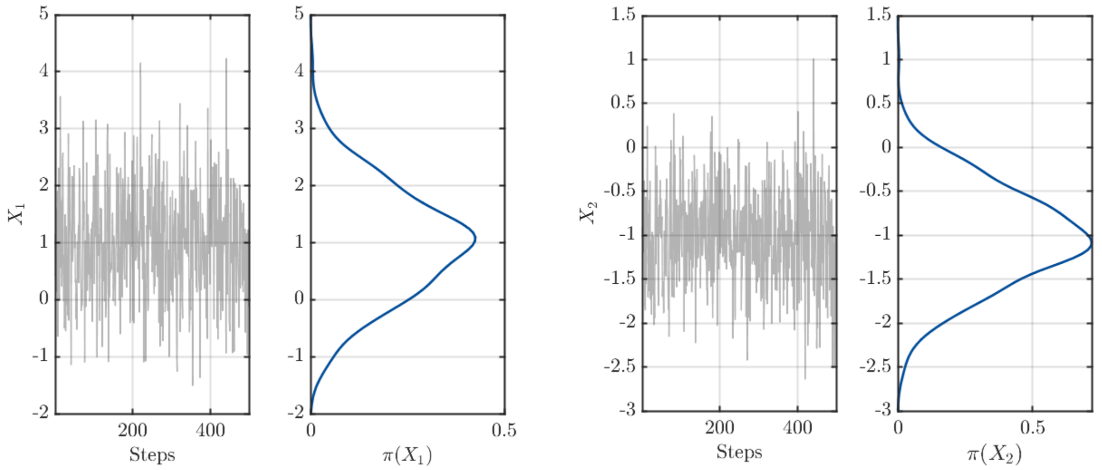


Figure 1: Two trace plots based on a 500×2 matrix.

5 Input

Table 1: `uq_traceplot(...)`

●	`X`	$N \times M \times C$ Double	Data matrix. If a 3-dimensional array is given, then for each trace plot, C separate lines are created.
☐	`AXES`	$M \times 2$ Cell array of `Axes` objects	`Axes` objects in which the trace plots will be created
☐	`NAME, VALUE`	name-value pair (See Table 2)	Additional options as name-value pairs

Table 2: uq_traceplot(..., NAME, VALUE)		
'labels'	$1 \times M$ Cell Array default: {'X_1','X_2',...,'X_M'}	Cell array of strings with labels used in the plots

6 Output

Table 3: PLOTAX = uq_traceplot(...)		
PLOTAX	$M \times 2$ Cell array of Axes objects	Axes objects in which the trace plots are created. In MATLAB R2014a or older, they are handles (*i.e.*, unique identifiers) to the Axes objects. The first and second columns correspond to the trace and kernel density plots for each data dimension, respectively.

uq_legend – Create a legend

1 Objective

Create a legend following the default formatting of UQLAB.

2 Description

uq_legend wraps the legend function in MATLAB and sets the UQLAB default formatting style on the resulting Legend object. The input arguments of uq_legend conform to the arguments of legend. The properties specifically set by uq_legend are listed in Table 1.

Table 1: uq_legend defaults

'Box'	'on'
'Color'	'white'
'EdgeColor'	'white'
'Interpreter'	'LaTeX'
'Location'	'best'

3 Syntax

```
uq_legend
uq_legend(...)
uq_legend(..., NAME, VALUE)
LGD = uq_legend(...)
```

uq_legend creates a legend on the current axes following the default formatting of UQLAB.

uq_legend(...) wraps the standard MATLAB legend function and sets the default formatting of UQLAB on the resulting Legend object. The input arguments to uq_legend conform to the input arguments of the MATLAB function.

uq_legend(..., NAME, VALUE) modifies the properties of the legend according to the specified NAME/VALUE pairs. The complete list of the pairs can be found in the documentation of the legend function.

`LGD = uq_legend(...)` returns the `Legend` object. In MATLAB R2014a or older, the function returns a handle to the legend (which is an `Axes` object). Use `LGD` to access and modify the properties of the legend after it has been created.

4 Examples

4.1 Add a legend to a plot with the default formatting

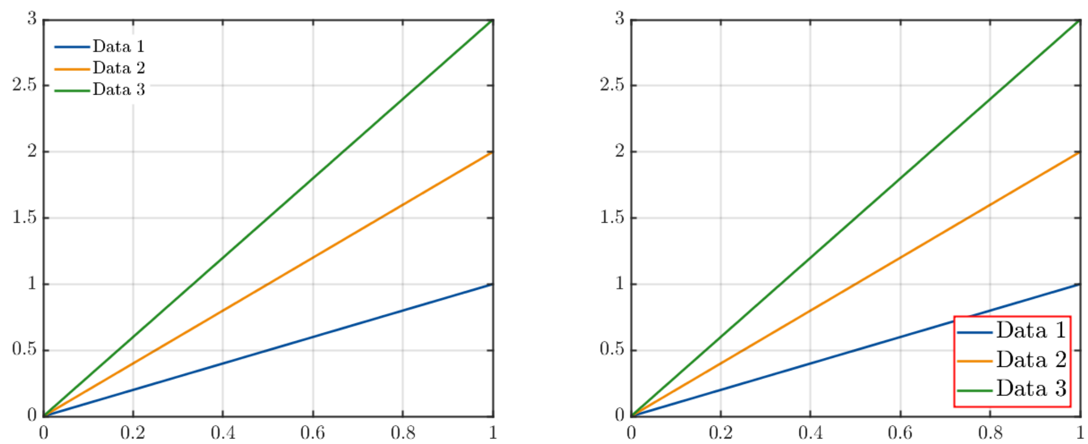
Below is an example how to create a legend with `uq_legend`. The resulting legend is shown in Figure 1(a).

```
x = repmat(linspace(0,1)',1,3);
y = x .* (1:3);
uq_plot(x,y)
uq_legend('Data 1', 'Data 2', 'Data 3')
```

4.2 Add a legend to a plot with a custom formatting

To customize the legend, a set of name-value pairs can be used. The default values listed in Table 1 can be overriden using the same approach. Below is an example of customizing a legend created using `uq_legend`. The resulting legend is shown in Figure 1(b).

```
x = repmat(linspace(0,1)',1,3);
y = x .* (1:3);
uq_plot(x,y)
uq_legend(...
  {'Data 1', 'Data 2', 'Data 3'},...
  'Box', 'on',...
  'EdgeColor', 'r',...
  'Location', 'southeast',...
  'FontSize', 20)
```



(a) Default formatting

(b) Custom formatting

Figure 1: Two plots with default and custom formatting styles for the legends.

5 Input

The input arguments for `uq_legend` are identical to the ones of the `legend` function in MATLAB. Refer to its documentation for details.

6 Output

Table 2: `H = uq_legend(...)`

`LGD`	`Legend` Object	Use the object to access and modify the properties of the legend. In MATLAB R2014a or older, the function returns a handle (*i.e.*, unique identifiers) to the legend, which is also an `Axes`.

7 Notes

- In MATLAB R2014a or older, the `'FontSize'` property of the legend has the same value as the `'FontSize'` property of the parent `Axes` object. In MATLAB R2014b or newer, the `'FontSize'` property of the legend is, by default, 90% of the the `'FontSize'` property of the parent `Axes` object.

uq_map – Map a sequence using a function

1 Objective

Evaluate a function on each element of an input sequence and return the evaluation results inside a cell array of the same length as the length of the input sequence.

2 Description

Given an input sequence and a function, uq_map evaluates the given function on each element of the sequence. In other words, uq_map defines a *mapping* from one sequence to another sequence using the given function (from here on in refered to as a *mapping function*). uq_map always returns a cell array of the same length as the input sequence; each element of the cell array contains the results of the function evaluations. In effect, the results of each function evaluation may have different data types. Figure 1 illustrates the process of mapping a sequence to another by uq_map.

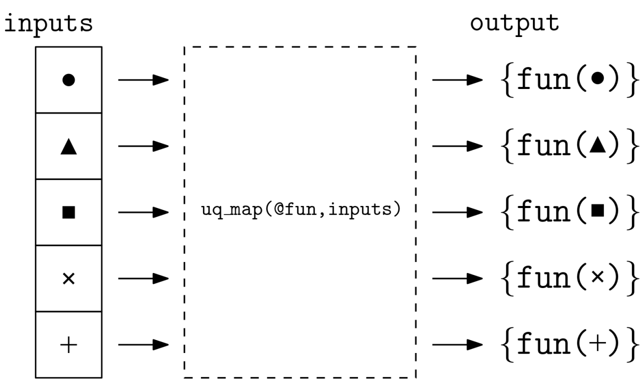


Figure 1: An illustration of a mapping process by uq_map.

Note: uq_map is a *dispatcher-aware* function. That is, the function is readily dispatchable to a remote machine and, when possible, executed in parallel using a DISPATCHER object. For details on using uq_map coupled with a DISPATCHER object, the reader is referred to the UQLAB User Manual – The HPC Dispatcher module, in particular, Appendix C (Wicaksono et al., 2021).

2.1 Types of mapping function

uq_map supports different types of mapping functions as its input, including MATLAB built-in functions, user-defined functions, anonymous functions, and system commands. Table 1 provides some remarks for each of these types.

Table 1: Supported functions for an input to uq_map.

Mapping function type	Remark
MATLAB built-in functions	All built-in functions are supported.
User-defined functions	The function m-file (and all its dependencies) must be available in the MATLAB path.
Anonymous functions	If an anonymous function is assigned to a variable, the variable is passed to uq_map without the preceding @.
System command	• System commands are specified as a string (*i.e.*, char array) and executed using system command of MATLAB. • Each mapping function evalution returns a struct that captures the exit status and terminal output from the command execution. • Valid system commands are specific to the operating system (OS).

The mapping function, with the exception of system command, is passed to uq_map as a function handle.

2.2 Types of sequence

uq_map takes a sequence represented by different MATLAB data types, including structure arrays, cell arrays, vectors, and matrices. The notion of *an element of the sequence* differs from type to type, and with an optional named argument, it can be modified as explained below.

Structure arrays are the most straightforward type of sequence to map using uq_map. This is illustrated in Figure 2 where each element of the structure array is used as input to the mapping function. Notice that the shape of the input structure array is preserved in the output, including in higher dimensions.

By default, when the input sequence is a cell array, taking an element out of it behaves the same way as in the case of structure array. The content (*i.e.*, value) of cell array element is passed to the mapping function. Being a cell array, the content of an element can be of any, even mixed, types.

However, with named argument 'ExpandCell' followed by a logical true or false, this

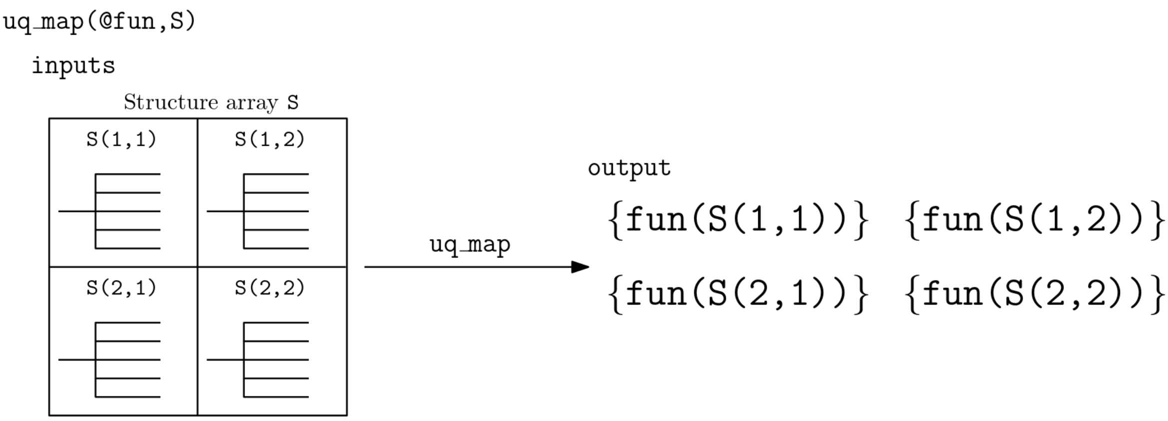


Figure 2: An illustration of a mapping process by uq_map on a structure array.

behavior can be changed. If 'ExpandCell' is set to true, then the content of the cell array element is expanded into a comma-separated list. This construction is useful if the input sequence has as its element an argument list for the mapping function. The difference between two settings of 'ExpandCell' is illustrated in Figure 3. Note once more that the shape of the input cell array is preserved in the output cell array.

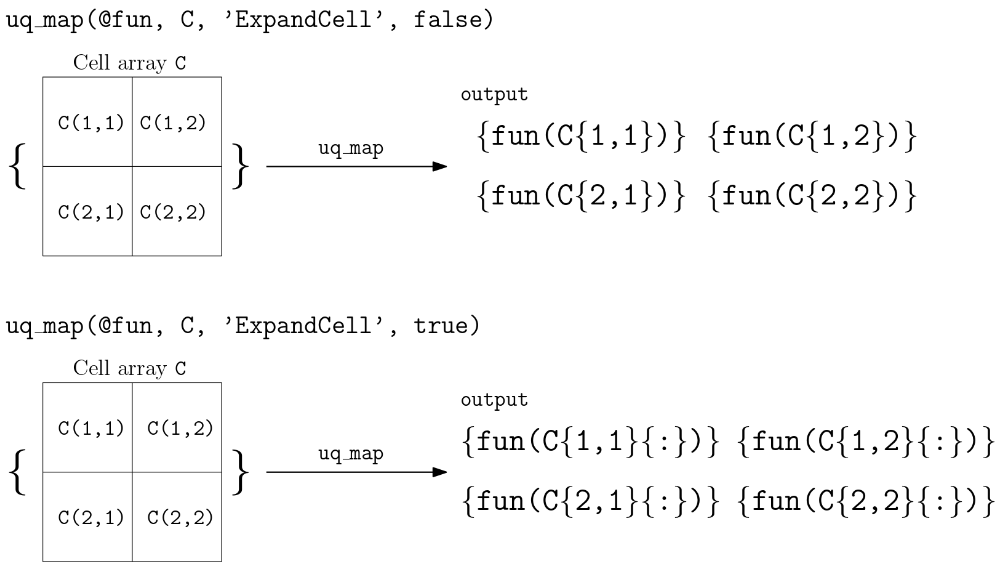


Figure 3: An illustration of a mapping process by uq_map on a cell array.

> **Note:** The named argument 'ExpandCell' is only relevant for a cell array as the input sequence uq_map.

When the input sequence is a numerical vector (or a matrix), then by default, each element of the vector (resp, matrix) is passed to the mapping function. This behavior can be changed using the named argument 'MatrixMapping' and it is particularly relevant if the input is a matrix. Three different values are possible:

- 'ByElements': Each element of the vector or matrix, *i.e.*, a scalar, is passed to the mapping function (the default).

- `'ByRows'`: Each row of the vector or matrix, *i.e.*, a row vector, is passed to the mapping function. If the input sequence is a row vector, then the sequence consists of only one element.
- `'ByColumns'`: Each column of the vector or matrix, *i.e.*, a column vector, is passed to the mapping function. If the input sequence is a column vector, then the sequence consists of only one element.

Figure 4 illustrates the different ways of taking an element from a matrix.

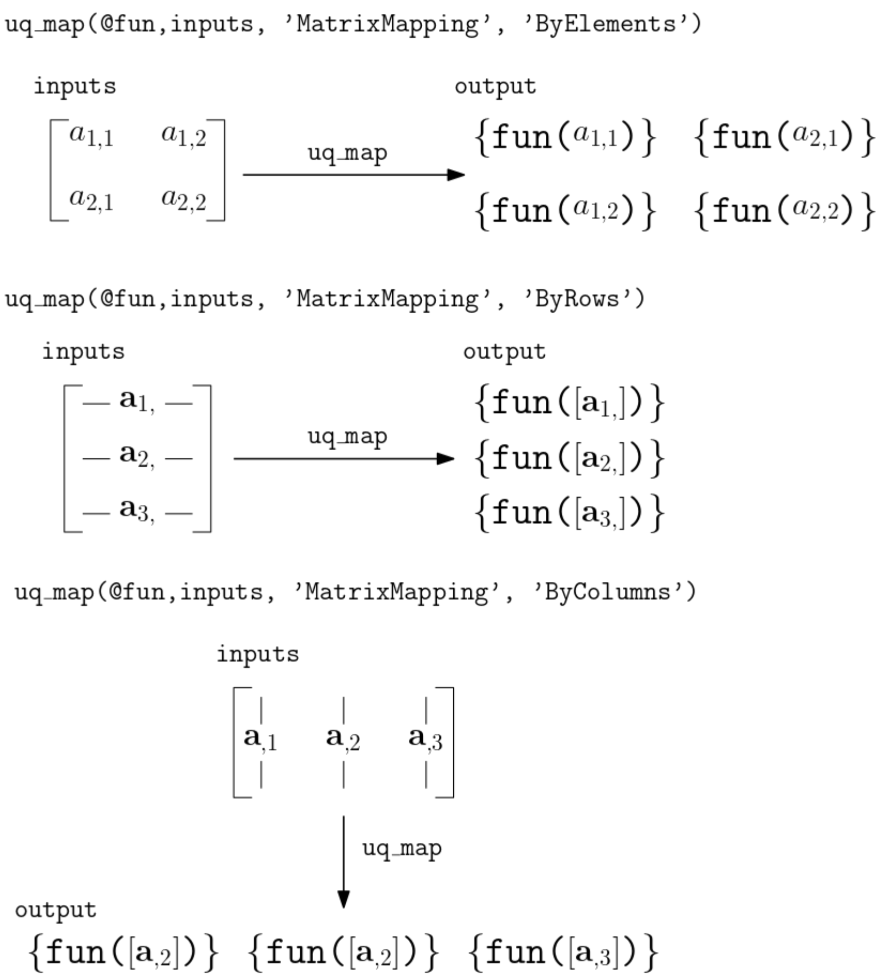


Figure 4: An illustration of a mapping process by `uq_map` on a matrix.

> **Note:** `uq_map` only supports numerical array up to 2 dimension (*i.e.*, a matrix). In this case, the input element would either be a scalar or a vector. To use an array of higher dimension as an element passed to the mapping function, use cell arrays instead.

2.3 Passing parameters

A argument of the mapping function may be passed as a *parameter*. While the mapping function is evaluated on each element of the input sequence, the value of a parameter argument will be kept as-is in each evaluation. An argument passed as a parameter is always positioned the last in the argument list of the mapping function. Figure 5 illustrates calling `uq_map` with a parameter.

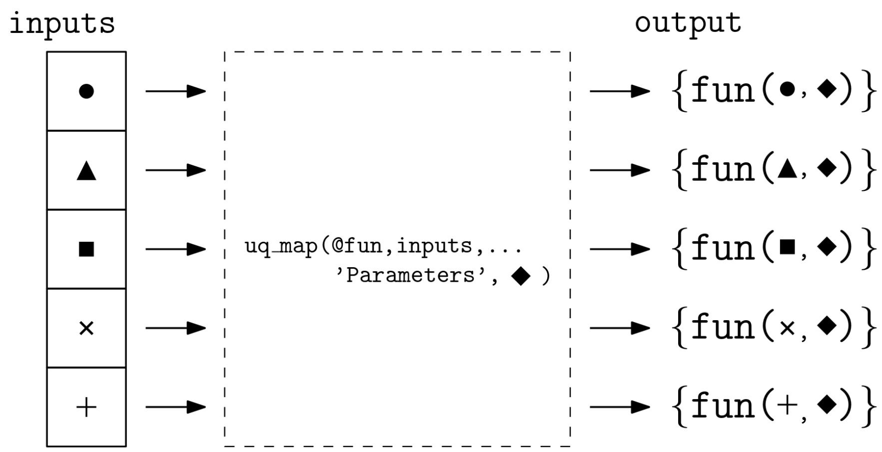


Figure 5: An illustration of a mapping process with a parameter.

A parameter is specified in `uq_map` using the named argument `'Parameters'` followed by the parameter. The parameter can take any MATLAB data types as long as it is a valid argument for the mapping function.

2.4 Mapping functions with multiple outputs

`uq_map` supports evaluating mapping functions that return multiple output arguments. If a mapping function returns multiple output arguments, `uq_map` can be called with multiple output arguments as well. The results of `uq_map` will then be N_{out} cell arrays, where N_{out} is the number of requested output arguments. These cell arrays contain the combined outputs from the mapping function evaluation on each element of the input sequence.

An example is provided in Section 4.5 to illustrate calling `uq_map` on a mapping function with multiple outputs.

2.5 Handling errors

By default, any error from evaluating a mapping function is automatically handled by returning `NaN` values; whose dimension is consistent with the specified number of output arguments. For a diagnostic purpose, this can be turned off by using the named argument `'ErrorHandler'` followed by a logical `false`. If error handling is turned off, any error thrown from a mapping function evaluation will causes `uq_map` to fail.

Moreover, a custom user-defined error handler may also be specified. If a mapping function throws an error, then the user-defined error handler catches the error and takes the steps specified in the handler. The error handler either must throw an error or return the same number of outputs as specified for the mapping function (in most cases `varargout` will suffice).

The user-defined error handle is a function, specified as a comma-separated pair consisting of `'ErrorHandler'` and a function handle. If the mapping function throws an error, then the error handler specified by `'ErrorHandler'` catches the error and takes the action specified in the function.

A user-defined error handler has the following signature:

```
function varargout = myErrorHandler(S,varargin)
```

The first input argument of the error handler is a structure with these fields:

- `identifier`: Error identifier
- `message`: Error message text
- `index`: linear index into the input sequence at which the mapping function threw the error

The remaining input arguments (a cell array) to the error handler are the input arguments for the call to the mapping function that made the function throws the error.

When an error is thrown, the error handler is called. Within this function, users then have access to the error identifier and message as well as the index of the sequence and the complete argument list that cause the error. It is up to the users to return this information for diagnostic purposes or some selected values to replace the return value of the mapping function return in the case of error.

2.6 System commands as a mapping function

Using system commands as a mapping function is a special use case of `uq_map`. In this case, a system command (represented as char array) is executed by MATLAB `system` command. The results of executing `system`, namely the command exit status and the command terminal output, are returned by `uq_map`.

The system command itself is typically composed from a template. In a command template, *replacement fields* are used as placeholders that later on will be replaced with the actual values from the input sequence; the resulting char array is then passed to MATLAB `system`.

When using system command as a mapping function, only 2-level nested cell arrays are supported as the input sequence. Each cell array element contains a list of values used to replace the placeholders in the command template.

A replacement field has the following syntax:

```
{<posID>[:<formatSpec>]}
```

The field, delimited by curly braces, contains two components:

1. `<posID>` specifies the position ID of the value from the cell array. Using position ID, a scalar integer, values from the input can be flexibly arranged.

2. `<formatSpec>` (optional) specifies how the value should be printed. It uses the standard MATLAB formatting operators. If not specified, the `formatSpec` for character vector (`%s`) is used.

Figure 6 provides an illustration how the replacement field works in practice when system command is specified as the mapping function to `uq_map`. The illustration shows how a single command works, but there is no limitation of how many commands are to be executed as long as the command char is composed properly.

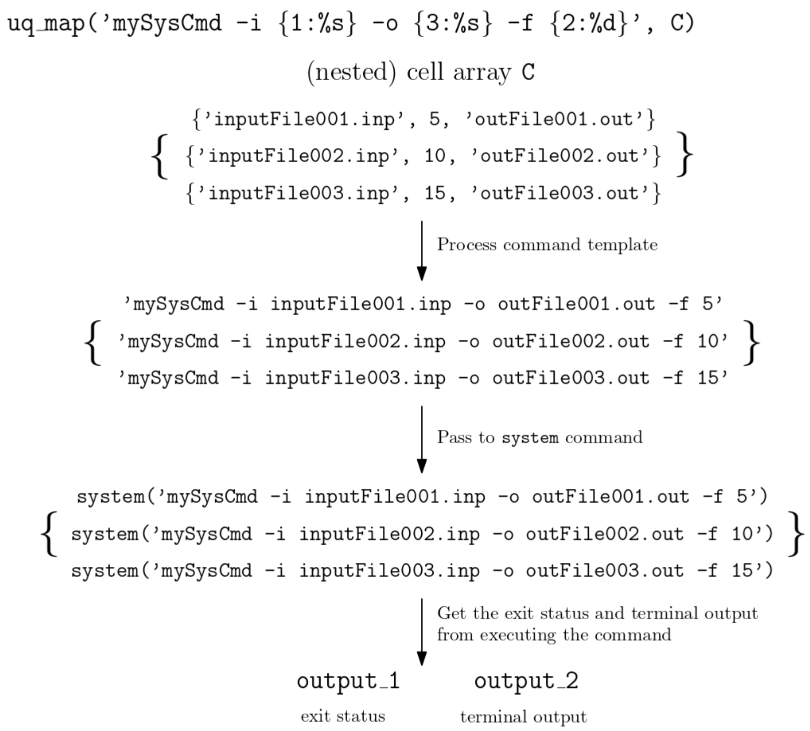


Figure 6: An illustration of a mapping process using a system command as the mapping function.

Note: While `uq_map` returns the exit status and the terminal output from executing the command, they are seldom the main purpose of executing the command. With respect to the user's MATLAB session, the side-effects (*e.g.*, executing external codes) of executing such command are often the main goal.

3 Syntax

```
OUTPUT = uq_map(FUN,INPUTS)
[OUTPUT_1,...,OUTPUT_NOUT] = uq_map(...)
[...] = uq_map(..., NAME, VALUE)
```

`OUTPUT = uq_map(FUN,INPUTS)` maps a sequence `INPUTS` to another sequence `OUTPUT` by evaluating a function handle `FUN` on each element of `INPUTS`. `OUTPUT` is a cell array with the number of elements the same as in `INPUTS`. If any evaluation of `FUN` on the elements of `INPUTS` fails, a `NaN` is returned.

`[OUTPUT_1,...OUTPUT_NOUT] = uq_map(FUN,INPUTS)` maps a sequence `INPUTS` using `FUN` to a set of cell arrays `OUTPUT_1,...,OUTPUT_NOUT` where `NOUT` is the number of requested outputs from `FUN`.

`[...] = uq_map(..., NAME, VALUE)` maps `INPUTS` by evaluating `FUN` on each of its elements with additional (optional) `NAME/VALUE` pairs (see Table 3).

4 Examples

4.1 Map a structure array

Create a structure array that contains random numbers:

```
rng(100,'twister')  % for reproducibility
for i = 1:4
  S(i).X = randn(1e3,1);
  S(i).Y = 1 + 2*randn(1e3,1);
end
```

Compute the mean of field `X` and field `Y` for each element of the structure array:

```
output = uq_map(@(s) [mean(s.X) mean(s.Y)],S)
```

```
output =

  1x4 cell array

    {1x2 double}    {1x2 double}    {1x2 double}    {1x2 double}
```

A custom function handle is created to combine the results of the MATLAB built-in functions `mean` of each field.

The result of the first element of the structure array is indeed a row vector:

```
output{1}
```

```
ans =

   -0.0192    0.9348
```

Notice that the size of the input array is the same as the output array.

4.2 Map a cell array

Create an anonymous function that creates a sample of 1000 points from normal distribution with parametrized mean and standard deviation:

```
rng(100,'twister')  % For reproducibility
fun = @(x) x{1} + x{2} * randn(1e3,1);
```

Create a cell array that contains the argument to the function handle:

```
C = {{0,1}; {1,2}; {2,3}};
```

Generate the samples from the three different arguments:

```
output = uq_map(fun,C)
```

```
output =

  3x1 cell array

  {1000x1 double}
  {1000x1 double}
  {1000x1 double}
```

Notice that if a handle is assigned to a variable, the variable name is passed directly to uq_map.

Alternatively, a function handle may be defined as follows:

```
fun = @(mu,sd) mu + sd * randn(1e3,1);
```

In this case, the element of the input cell array can be directly expanded to a comma-separated list and passed as an argument list into the mapping function:

```
output = uq_map(fun, C, 'ExpandCell', true)
```

```
output =

  3x1 cell array

  {1000x1 double}
  {1000x1 double}
  {1000x1 double}
```

The results are identical. To verify the samples, compute the mean:

```
uq_map(@mean,C)
```

```
ans =
```

```
  3x1 cell array

    {[0.0604]}
    {[0.9896]}
    {[2.0411]}
```

and the standard deviation:

```
uq_map(@std,C)
```

```
ans =

  3x1 cell array

    {[1.0003]}
    {[1.9727]}
    {[2.8943]}
```

4.3 Map a matrix

Create a 4-by-3 random sample from a normal distribution:

```
A = rand(4,3);
```

Get the size of the each element using the `size` function:

```
output = uq_map(@size,A)
```

```
output =

  4x3 cell array

    {1x2 double}    {1x2 double}    {1x2 double}
    {1x2 double}    {1x2 double}    {1x2 double}
    {1x2 double}    {1x2 double}    {1x2 double}
    {1x2 double}    {1x2 double}    {1x2 double}
```

By default, mapping a matrix uses each element of a matrix as an input to the mapping function. Therefore, the output of `size` shows that the element is indeed a scalar:

```
output{1,1}
```

```
ans =

  1     1
```

By using `'ByRows'` for the named argument `'MatrixMapping'`, the elements will be taken from the rows of the matrix:

```
uq_map(@size, A, 'MatrixMapping', 'ByRows')
```

```
output =
```

```
  4x1 cell array

    {1x2 double}
    {1x2 double}
    {1x2 double}
    {1x2 double}
```

The output of `size` shows that the element is a row vector:

```
output{1}
```

```
ans =

  1     3
```

By using `'ByColumns'` for the named argument `'MatrixMapping'`, the elements will be taken from the columns of the matrix:

```
uq_map(@size, A, 'MatrixMapping', 'ByColumns')
```

```
output =

  1x3 cell array

    {1x2 double}    {1x2 double}    {1x2 double}
```

The output of `size` shows that the element is a column vector:

```
output{1}
```

```
ans =

  4    1
```

4.4 Map a sequence using a parameter

A parameter can be passed to `uq_map` and, in turn, will be passed to each call to the mapping function as a parameter; its value is the same for each call.

Create a cell array of 3 random matrices (each matrix has different number of rows):

```
for i = 1:3
  C{i} = rand(i,1e3);
end
```

Compute the mean of each row of the matrices (the second argument of `mean` is set to 2):

```
uq_map(@mean, C, 'Parameters', 2)
```

```
output =
```

```
  3x1 cell array

    {[   0.5043]}
    {2x1 double}
    {3x1 double}
```

4.5 Map a sequence using a function with multiple outputs

Create a cell array of random matrices:

```
rng(100,'twister')  % for reproducibility
for i = 1:3
  C{i} = rand(5,5);
end
```

Compute the singular value decomposition (SVD) of each of these matrices using the `svd` function (the `svd` function returns multiple outputs) and get all its outputs (*i.e.*, the matrices U, S, and V):

```
[U,S,V] = uq_map(@svd, C)
```

```
U =

  1x3 cell array

    {5x5 double}    {5x5 double}    {5x5 double}

S =

  1x3 cell array

    {5x5 double}    {5x5 double}    {5x5 double}

V =

  1x3 cell array

    {5x5 double}    {5x5 double}    {5x5 double}
```

There are three output arrays that contain the combined outputs from the mapping function evaluation on each element of the input sequence. For instance, the matrix stored in `S{2}` is the matrix `S` from `svd(C2)`:

```
S{1}
```

```
  ans =

  2.4977         0         0         0         0
       0    1.0610         0         0         0
       0         0    0.6075         0         0
```

```
         0          0          0     0.3753          0
         0          0          0          0     0.3017
```

4.6 Handle an error

Create an anonymous function of two arguments:

```
fun = @(x,y) x + y;
```

Evaluate the function on a 2-element cell array using uq_map:

```
output = uq_map(fun,{1;2})
```

Because the function requires two inputs but each element of the sequence provides only one, the evaluation fails. By default, a failed evaluation returns NaN:

```
output =

  2x1 cell array

    {[NaN]}
    {[NaN]}
```

If the error handler is turned off, uq_map will throw an error instead:

```
output = uq_map(fun, {1;2}, 'ErrorHandler', false)
```

```
Not enough input arguments.

Error in @(x,y)x+y

...
```

A user-defined function handler can also be specified. The handler must be saved as a function in a separate m-file. For example:

```
function output = myErrorHandler(S,varargin)

output.ME = S;
output.Input = varargin;

end
```

The above error handler returns the error identifier and messages as well as the input argument on which the mapping function throws an error. These are useful for diagnostic purposes.

Calling uq_map with the user-defined error handler:

```
output = uq_map(fun, {{1,2};{3,4}}, 'ErrorHandler', @myErrorHandler)
```

```
output =

  2x1 cell array

    {1x1 struct}
    {1x1 struct}
```

The details on the error for, for example, the first evaluation:

```
output{1}.ME
```

```
ans =

  struct with fields:

    identifier: 'MATLAB:minrhs'
       message: 'Not enough input arguments.'
         index: 1
```

4.7 Map a system command

Execute `echo` command (works both on Windows and Linux operating systems) on a set of char arrays:

```
output = uq_map('echo {1}', {'Hello'; 'Goodbye'})
```

```
output1 =

  2x1 cell array

    {[0]}
    {[0]}

output2 =

  2x1 cell array

    {'Hello'   }
    {'Goodbye'}
```

The exit status and the terminal output of the command execution are stored in the first and second output, respectively.

5 Input

Table 2: uq_map(FUN, INPUTS, NAME, VALUE)

●	FUN	Function handle or char array	Mapping function, used to evaluate (i.e., *map*) each element of INPUTS. See Table 1 for some remarks on the supported types of function.
●	INPUTS	Cell array, structure array, or matrix	Sequence whose elements are passed as inputs into FUN.
☐	NAME, VALUE	name-value pair (See Table 3)	Additional options as name-value pairs.

Table 3: uq_map(..., NAME, VALUE)

'Parameters'	Any MATLAB data types default: 'none'	Parameters passed to FUN as the last positional argument. Parameters are kept constant for each call to FUN. The type of parameters depends on FUN. See Section 2.3 for details.
'ExpandCell'	logical default: false	Flag to expand the content of a cell into a comma-separated list. This is useful if the contents of cell element are to be pass as a list of arguments to FUN. See Section 2.2 for an illustration.
'MatrixMapping'	String default: 'ByElements'	Way to take an element from a matrix. Possible values: • 'ByElements': each element is an individual scalar from the matrix. • 'ByRows': each element is a row vector corresponds to each row of the matrix. • 'ByColumns': each element is a column vector corresponds to each column of the matrix. See Section 2.2 for an illustration.

Continued on next page

Table 3–continued from previous page

'ErrorHandler'	logical or function handle default: true	Way to handle an error from evaluating the mapping function on elements of the input sequence. Possible values: • true: Any error automatically returns NaN. • false: Errors are not handled; any error causes uq_map to throw an error. • a function handle: a user-defined error handler. See Section 2.5 for details.

6 Output

Table 4: [OUTPUT_1,...,OUTPUT_NOUT] = uq_map(...)

OUTPUT_1,...	Cell array(s)	Results of evaluating FUN on each element of INPUTS. The number of elements in OUTPUT is always the same as the number of elements in INPUTS. If FUN supports multiple output arguments, uq_map may also be called with multiple output arguments. The results are multiple cell arrays, each of which contains each of the outputs from FUN evaluation on the input sequence. If a system command is used as FUN, two output arrays may be requested. Each of the output arrays has the same size as INPUTS. The first and second output array contains the exit status and terminal output from executing the command, respectively. (see Section 2.6).

7 Notes

- uq_map is a dispatcher-aware function thus readily dispatchable to a remote machine. However, if a DISPATCHER object is to be used, it is recommended to test uq_map locally (*i.e.*, without the DISPATCHER) on, possibly, limited set of the inputs to check if the call is valid.

References

Arnold, D. V. and N. Hansen (2010). Active covariance matrix adaptation for the (1+1)-CMA-ES. In M. Pelikan and J. Branke (Eds.), *Proc. of the Genetic and Evolutionary Computation Conference 2010 (GECCO 2010)*, pp. 385–392. 35, 43

Arnold, D. V. and N. Hansen (2012). A (1+1)-CMA-ES for constrained optimisation. In T. Soule and J. H. Moore (Eds.), *Proc. of the Genetic and Evolutionary Computation Conference 2012 (GECCO 2012)*, pp. 297–304. 43, 49

Chapra, S. C. and R. P. Canale (2015). *Numerical methods for engineers*, Chapter Numerical Differentiation, pp. 655–672. New York, USA: McGraw Hill Education. 8

Cherkassky, V. and F. Mulier (2007). *Learning from data: concepts, theory, and methods*. Wiley. 2, 52

Friedman, J. H., J. L. Bentley, and R. A. Finkel (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS) 3*(3), 209–226. 60

Hansen, N. (2001). The CMA evolution strategy: A tutorial. Technical report, Institut National de Recherche en Informatique et en Automatique (INRIA), France. 26, 32, 34

Hansen, N. and S. Kern (2004). Evaluating the CMA evolution strategy on multimodal test functions. *Evolutionary Computation 9*(2), 282–221. 26

Hansen, N. and A. Ostermeier (2001). Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation 9*(2), 159–195. 26, 33

Igel, C., T. Suttorp, and N. Hansen (2006). A computational efficient covariance matrix update and a (1+1)-CMA for evolution strategies. In *Proc. of the Genetic and Evolutionary Computation Conference 2006 (GECCO 2006)*, pp. 453–460. 35

Kroese, D. P., S. Porotsky, and R. Y. Rubinstein (2006). The cross-entropy method for continuous multi-extremal optimization. *Methodology and Computing in Applied Probability 8*, 383–407. 19

Lataniotis, C., D. Wicaksono, S. Marelli, and B. Sudret (2021). UQLab user manual – Kriging. Technical report, Chair of Risk, Safety & Uncertainty Quantification, ETH Zurich. Report # UQLab-V1.4-105. 2

Lloyd, S. (1982). Least squares quantization in PCM. *IEEE transactions on information theory 28*(2), 129–137. 60

Marelli, S., N. Lüthen, and B. Sudret (2021). UQLab user manual – Polynomial chaos expansions. Technical report, Chair of Risk, Safety & Uncertainty Quantification, ETH Zurich. Report # UQLab-V1.4-104. 2

Moustapha, M., C. Lataniotis, S. Marelli, and B. Sudret (2021a). UQLab user manual – Support vector machines for classification. Technical report, Chair of Risk, Safety & Uncertainty Quantification, ETH Zurich. Report # UQLab-V1.4-112. 25

Moustapha, M., C. Lataniotis, S. Marelli, and B. Sudret (2021b). UQLab user manual – Support vector machines for regression. Technical report, Chair of Risk, Safety & Uncertainty Quantification, ETH Zurich. Report # UQLab-V1.4-111. 25

Rasmussen, C. and C. Williams (2006). *Gaussian processes for machine learning*. Adaptive computation and machine learning. Cambridge, Massachusetts: MIT Press. 52, 54

Rubinstein, R. and W. Davidson (1999). The cross-entropy method for combinatorial and continuous optimization. *Methodology and Computing in Applied Probability 1*(2), 127–190. 19

Rubinstein, R. Y. (1997). Optimization of computer simulation models with rare events. *European Journal of Operational Research 99*, 89–112. 19

Sacks, J., W. J. Welch, T. J. Mitchell, and H. P. Wynn (1989). Design and analysis of computer experiments. *Statistical Science 4*, 409–435. 54

Scott, D. W. (2010). Scott's rule. *Wiley Interdisciplinary Reviews: Computational Statistics 2*(4), 497–502. 75

Shawe-Taylor, J. and N. Cristianini (2004). *Kernel methods for pattern analysis*, Chapter Properties of kernels, pp. 47–84. Cambridge, UK: Cambridge University Press. 52

Suttorp, T., N. Hansen, and C. Igel (2009). Efficient covariance matrix update for variable metric evolution strategies. *Machine learning 75*(2), 167–197. 35, 40, 49

Vapnik, V. (1995). *The Nature of Statistical Learning Theory*. Springer-Verlag, New York. 52

Wicaksono, D., C. Lamas, C. Lataniotis, S. Marelli, and B. Sudret (2021). UQLab user manual – the HPC Dispatcher module. Technical report, Chair of Risk, Safety & Uncertainty Quantification, ETH Zurich. Report # UQLab-V1.4-116. 94