UQLab User Manual
The HPC Dispatcher Module

D. Wicaksono, C. Lataniotis, 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

D. Wicaksono, C. Lataniotis, S. Marelli, B. Sudret, UQLab user manual – The HPC dispatcher module, Report UQLab-V2.0-116, Chair of Risk, Safety and Uncertainty Quantification, ETH Zurich, Switzerland, 2022

BibTeX entry

```
@TechReport{UQdoc_20_116,
author = {Wicaksono, D. and Lataniotis, C. and Marelli, S. and Sudret, B.},
title = {{UQLab user manual -- The HPC Dispatcher module}},
institution = {Chair of Risk, Safety and Uncertainty Quantification, ETH Zurich,
Switzerland},
year = {2022},
note = {Report UQLab-V2.0-116}
}
```

List of contributors:

C. Lamas	Initial implementation of the synchronous dispatching for `uq_evalModel`

Document Data Sheet

Document Ref.	UQLab-V2.0-116
Title:	UQLab user manual – The HPC dispatcher module
Authors:	D. Wicaksono, C. Lataniotis, 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	Initial release

Abstract

The high-performance computing (HPC) DISPATCHER module is a UQLAB support module that connects UQLAB to common distributed computing or HPC infrastructure. The module provides a consistent interface for users to interactively set up, submit, keep track of, and retrieve the results of remote computations directly from within a UQLAB session.

This guide is an extensive manual of the HPC DISPATCHER module and divided into three parts:

- A short introduction to the main concepts and terminologies of distributed computing and dispatched computations with the DISPATCHER module;
- A detailed example-based user guide with explanations and examples of most of the available options and functionalities of the module, from setting up the DISPATCHER module to retrieving the results of dispatched computations from a remote machine;
- A comprehensive reference list of all the available options and functionalities of the module.

Keywords: UQLAB, high-performance computing, HPC, distributed computing, dispatched computation

Contents

Chapter 1

Introduction

Distributed computing resources, such as high-performance computing (HPC) clusters, give users possibilities to scale up, speed up, or offload (thus freeing up their laptop or desktop computers) their UQLab computations. However, a typical workflow of using such resources, from submitting the computations to retrieving the results, can be challenging for many UQLab users.

The HPC Dispatcher module allows one to connect UQLab to common distributed computing resources, providing a consistent programmatic interface to set up, submit, and retrieve parallel computational jobs directly from within UQLab. This seamless integration is designed to significantly enhance the user experience in carrying out expensive computations using distributed computing resources.

This user manual presents the HPC Dispatcher module of UQLab. This chapter introduces the relevant key terminologies of distributed computing. It then describes the workflow of dispatching a computation using the module, which slightly differs from a typical local computation with which users are familiar. Chapter 2 then details the setup of a Dispatcher object and the module's usage in several different scenarios, from simple UQLab model evaluation to more complex parametric studies. Finally, Chapter 3 provides a comprehensive list of the available commands and options in the Dispatcher module.

1.1 Motivation

While most UQLab users are able to carry out their uncertainty quantification (UQ) analyses on their personal computers (PC, be it laptops or desktops), there are several situations in which such computing resources may be limiting:

- *Long-running computations*: Though users might be willing to wait for their computations to finish, carrying out these computations on the PC might render their computer unusable for an extended period.
- *Exceptional resources (memory, CPU, storage space) requirements*: Some computations

may be so large that they are impractical or even impossible to carry out on a PC. A steep memory requirement is especially restrictive; while a long-running computation would eventually be finished, a memory restriction forbids the computation even to start.

- *Compatibility and licensing issues*: Computational models in a UQ analysis may be based on a simulation code that runs only on Linux operating systems. Furthermore, such a code might have a special licensing requirement not available on the PC of the user.

These are indeed the situations in which remote distributed computing resources (or simply, *remote machines*) might help. Typically, these machines:

- are highly available round the clock;
- have a large number of central processing units (CPUs) as well as a large amount of memory and storage space, well above what is available on standard commercial PC;
- come with shared license software installed and run a Linux operating system.

Remote machines are *remote* in the sense that users may not have direct physical access to them and these machines can be located anywhere in the world. Remote machines include a wide array of computing resources, from a single server to distributed high-performance computing (HPC) clusters (Figure 1). The latter refers to a distributed system in which multiple servers (or *nodes*) are interconnected by a high-performance network. Such a system is designed to handle large numbers of floating point operations per second (FLOPS with high data volumes, a typical situation in scientific computing. In this manual, remote machines may refer to any of these resources.

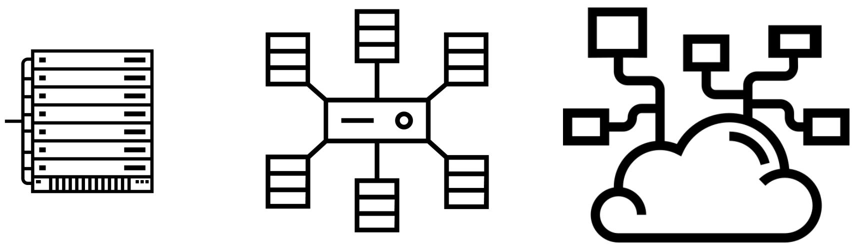


Figure 1: Various forms of distributed computing resources: remote server, HPC clusters, and cloud clusters.

However, using these machines to carry out a UQLAB computation may pose some difficulties to many UQLAB users. A large part of those difficulties stems from the fact that these machines are *remote* and (mostly) *shared* resources.

Being remote resources, a user interacts with a remote machine from their PC over the network using the encrypted Secure Shell (SSH) protocol, a protocol used to log in from one computer to another securely. Using this protocol, the user can issue commands from their

PC to the remote machine and transfer data (*i.e.*, files) between the PC and the remote machine (Figure 2). As opposed to a typical workflow on a PC, a computation is executed on the remote machine, and its results will not readily be available on the user's PC upon conclusion.

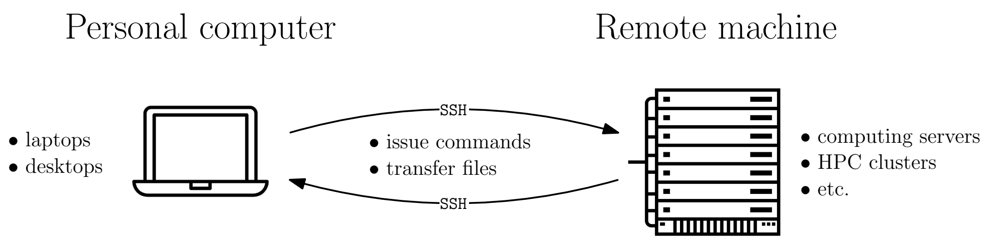


Figure 2: A user interacts with a remote machine from their PC over the network using the SSH protocol to issue commands and transfer files.

Being shared resources, remote machines (especially HPC clusters) typically employs a *job scheduler* (or simply, *scheduler*), a program to manage the computational tasks (called *jobs*) submitted by multiple users. A scheduler is responsible for, among other things, managing the job queue (*i.e.*, deciding which jobs to run and when) and ensuring that each job is executed with the resources it needs. As opposed to a typical workflow on a PC, when a scheduler is employed, a computation can only be scheduled and submitted to the remote machines. Its execution is not immediate and subject to the scheduler's policy. Furthermore, users do not have an interactive access to this scheduled computation. Consequently, the results must first be stored in files to be retrieved later.

In practice, for users to carry out their UQLAB computations on a remote machine, they commonly need to:

1. Log in to a remote machine using a command-line interface.
2. Modify their original computation script; the computation results must be saved in files to be retrieved later. If the computation can be parallelized, the script may need further—and non-trivial—modifications to benefit from a parallel execution.
3. Prepare a job script. A job script includes the specification of the computational requirement for the job (*e.g.*, the maximum duration, the number of CPUs, the memory size)
4. Submit the job to the scheduler on the remote machine.
5. Query the status of the job execution and wait until it is finished.
6. Once finished, bring the results back to the local machine (*i.e.*, PC).
7. Carry out the analysis of results on the local machine.

These steps, which have to be repeated for different computations, may prove challenging for many UQLAB users to benefit from the remote machine for some of their UQ analyses.

1.2 The HPC Dispatcher module

The HPC Dispatcher module attempts to ease the difficulties of using remote machines for UQLab computations by providing a user-friendly interface between the user's personal computer (PC) and a remote machine. The purpose of the module is, first and foremost, to *dispatch*[1] a computation to the remote machine. A user dispatches the computation from a Matlab/UQLab session running on their PC. In the rest of this manual, the Matlab/UQLab session and the PC it is running on are referred to as the *local* UQLab *session* and the *local client*, respectively. The dispatched computation will be executed on the remote machine instead of the local client. Figure 3 provides a summary of the different aspects of dispatching a computation to a remote machine. Each of these aspects are explained further below.

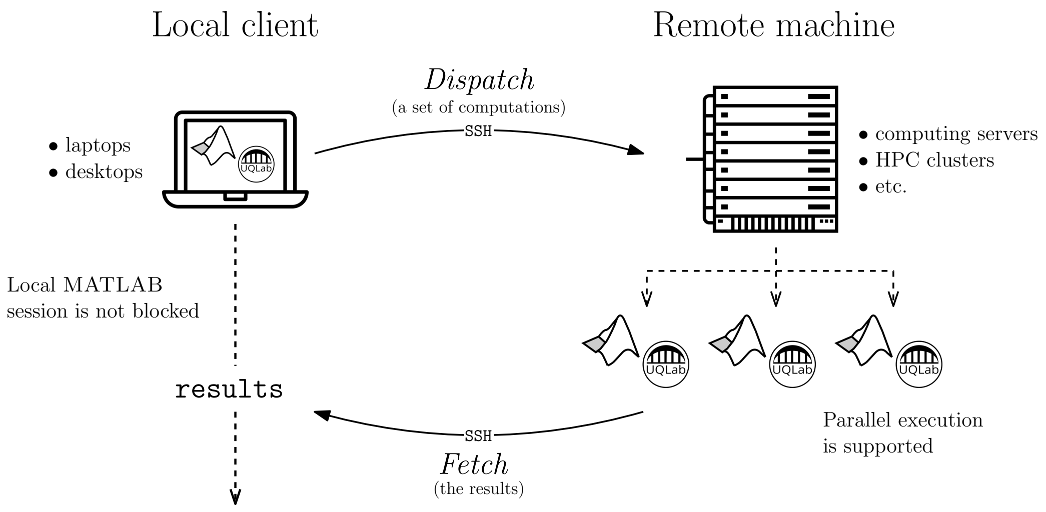


Figure 3: All the aspects of dispatching a computation using the Dispatcher module. A computation is *dispatched* from a local UQLab session running on a *local client* to a *remote machine*. The computation in the remote machine happens, in principle, *asynchronously*. The local UQLab session is not blocked during the execution; users are free to carry out other tasks. Once the execution on the remote machine is finished, users need to *fetch* the results back to their local UQLab session. Synchronization between the local UQLab session and the remote machine is possible, and is default behavior. Communications between the client and the remote machines are carried out over the network using the encrypted SSH protocol.

When a user dispatches their computations to the remote machine, in reality they *submit* the computation for execution to the scheduler on the remote machine. Once dispatched, the computations will be executed *asynchronously*. Its execution and eventual conclusion on the remote machines are independent of the local UQLab session. This execution mode also means that the execution does not need to block the local UQLab session; users are free to carry out other tasks in the session (including dispatching a new set of computations). This mode contrasts with the *synchronous* execution typical of local interactive session, in which the computation is executed immediately and the control is returned to the user only after the computation is finished.

[1]*to dispatch* is interchangeable with *to offload* or *to outsource*.

Note: The execution of dispatched computations, however, can still be synchronized. In other words, they are forced to behave as if they are executed synchronously. Indeed, this is the default behavior of dispatched computations to give the user experience of "local and synchronous" execution. Section 1.3.4 further explains the two different modes of dispatched computations.

Another implication of the remote and asynchronous execution mode is that, once the computation is finished, the results of a dispatched computation will not be automatically available in the local UQLAB session. The results remain stored in the remote machine, and users may need to explicitly *fetch* them back to the local client. Dispatching a computation and fetching the results present a slightly different workflow from the usual local computation, in which results of the computation are readily available in the local MATLAB session right after the execution is done. Section 1.3 details this workflow further.

All of the above computing resources—local and remote—also happen nowadays to be *parallel computing resources*. That is, they have multiple processors and can execute certain computations (or parts of that computation) simultaneously in parallel. Although this is not a requirement for dispatching a computation, parallel execution may speed up the computation. Whether a computation can benefit from parallel computing resources or not depends on the nature of the computation itself and on the overall computing infrastructure. When appropriate, the DISPATCHER module supports the parallel execution of a dispatched computation.

1.3 The HPC DISPATCHER workflow

To have a more detailed overview on what the DISPATCHER module does and how its usage workflow differs from a typical local computing, Figure 4 provides an illustration when a dispatcher-aware command is executed with and without a DISPATCHER object available in the current UQLAB session.

The main task of the DISPATCHER module is to handle the authentication and communication with remote machines as well as the bookkeeping of all the dispatched computations executed on the remote machines. In practical terms, when a computation is dispatched, the DISPATCHER module is responsible for (Figure 4):

- preparing a dispatch package which includes the relevant data and scripts to be executed on the remote machine;
- sending the package over the network using the encrypted SSH protocol;
- organizing the execution of the function on the remote machine;
- fetching the results back to the local UQLAB session.

The DISPATCHER module, along with the available dispatcher-aware commands (Section 1.3.1), hides many implementation details of dispatching a computation to a remote

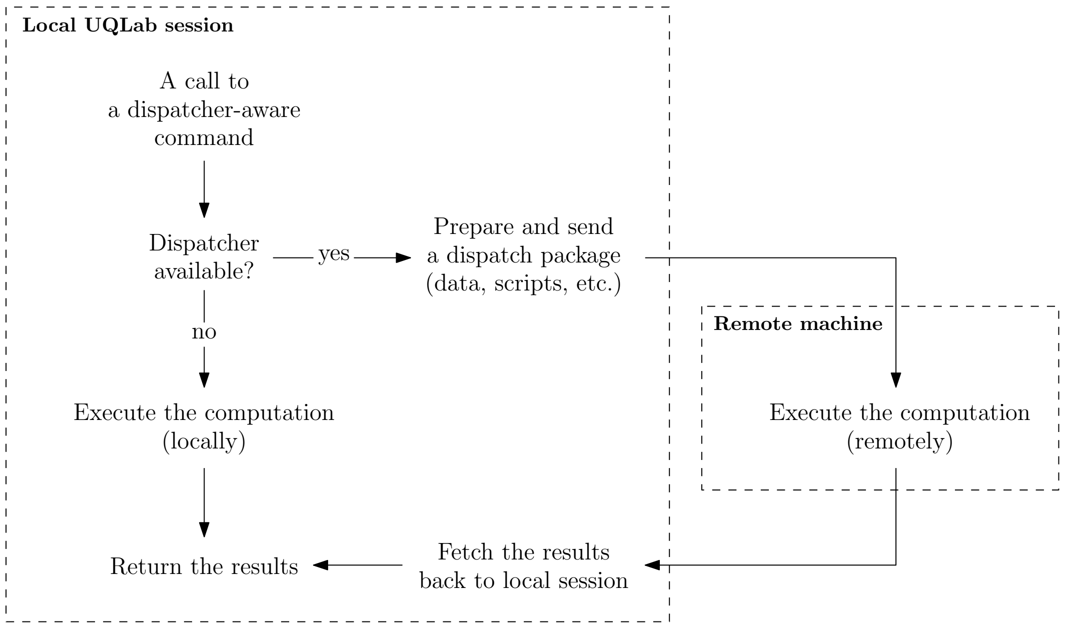

Local UQLab session
A call to a dispatcher-aware command
Dispatcher available?
yes
no
Prepare and send a dispatch package (data, scripts, etc.)
Remote machine
Execute the computation (locally)
Execute the computation (remotely)
Return the results
Fetch the results back to local session

Figure 4: A simplified view of the DISPATCHER module workflow. The connection to the remote machine is made using the encrypted SSH protocol. By default, the results of the remote execution persist in the remote machine.

machine and fetching the results back. There are relatively few new concepts for users to learn in a typical workflow involving a dispatched computation. The essential concepts of dispatching a computation from the user's point of view are discussed below.

1.3.1 Dispatcher-aware commands

The DISPATCHER module provides additional functionalities to the so-called *dispatcher-aware* commands of UQLAB. Dispatcher-aware commands are specifically developed to be executed in distributed computing resources and, possibly, in parallel. In the current incarnation of the DISPATCHER module, two commands, covering a wide array of functionalities, are dispatcher-aware out-of-the-box: `uq_evalModel` and `uq_map`.

Note: Not all the available functionalities of UQLAB are dispatcher-aware; only the dispatcher-aware commands can be dispatched. It is possible, however, to write custom dispatcher-aware functions.

UQLAB users are most likely familiar with `uq_evalModel`, a function to evaluate any UQLAB MODELS on a given set of input parameters. If used in conjunction with the DISPATCHER module, this evaluation can be dispatched to a remote machine and evaluated in parallel using multiple processors. All UQLAB MODELS, including UQLINK MODELS, can be dispatched. Section 2.3 provides usage examples of dispatching a MODEL evaluation to a remote machine.

The second dispatcher-aware command is `uq_map`, a function to evaluate a given arbitrary function on a given input. When the input is given as a sequence (*e.g.*, a cell array, a struct array, a matrix), the given function (*i.e.*, the *mapping* function) are evaluated for each element of the input sequence. uq_map returns the results from each evaluation; the results always have the same number of elements as in the input sequence. Using uq_map is an alternative to using a for-loop for the same evaluations; it evaluates the mapping function on a sequence of inputs one element at a time[2].

uq_map is a generic UQLab function and it works with or without the Dispatcher module. When used in conjunction with the Dispatcher module, however, the function evaluations can be dispatched to a remote machine and executed in parallel[3]. This command is particularly useful to dispatch user-defined functions that wrap around other UQLab functionalities to be executed on the remote machine(s). Section 2.4 provides an example of using uq_map coupled with the Dispatcher module to carry out a parametric study.

Note: The reader is referred to the uq_map chapter of the UQLib User Manual (Moustapha et al., 2021) for a reference on using uq_map without the Dispatcher module. In fact, some familiarity with uq_map before dispatching it to the remote machine is recommended.

1.3.2 Dispatching a computation

To dispatch a computation with any of the dispatcher-aware commands, a user must first configure and create a Dispatcher object. Once the Dispatcher object has been successfully created, the user dispatches the computation by calling a dispatcher-aware command.

This workflow is shown in Figure 5, along with an illustration of what happens behind the scenes. It shows that although the Dispatcher object performs many intermediate tasks to dispatch a computation to a remote machine, only few steps are required from the user.

After the computation is dispatched, an updated Dispatcher object is returned to the user regardless of the remote job execution outcome. As mentioned in Section 1.2, while the remote job execution, in principle, happens asynchronously to the local UQLab session, it is synchronized with the local UQLab session (see Section 1.3.4 for details).

1.3.3 Keeping track of a dispatched computation

Once the computation is dispatched, its execution happens on the remote machine, and the final results will also be stored in there. As shown in Figure 5, the notion of Job is central to the Dispatcher module. In UQLab/Dispatcher terminology, a Job refers to a distinct computation executed on the remote machine(s). Users keep track of dispatched computations by managing these Jobs from their local UQLab session.

[2]As a side note, a map function is a functional programming style's take on specific programmatic tasks that typically involve for-loops. uq_map borrows its name from a similar function available in several other programming languages, including R and Python.

[3]A similar parallelization can be also achieved by using a parfor-loop, a Matlab programming construct that requires a special toolbox.

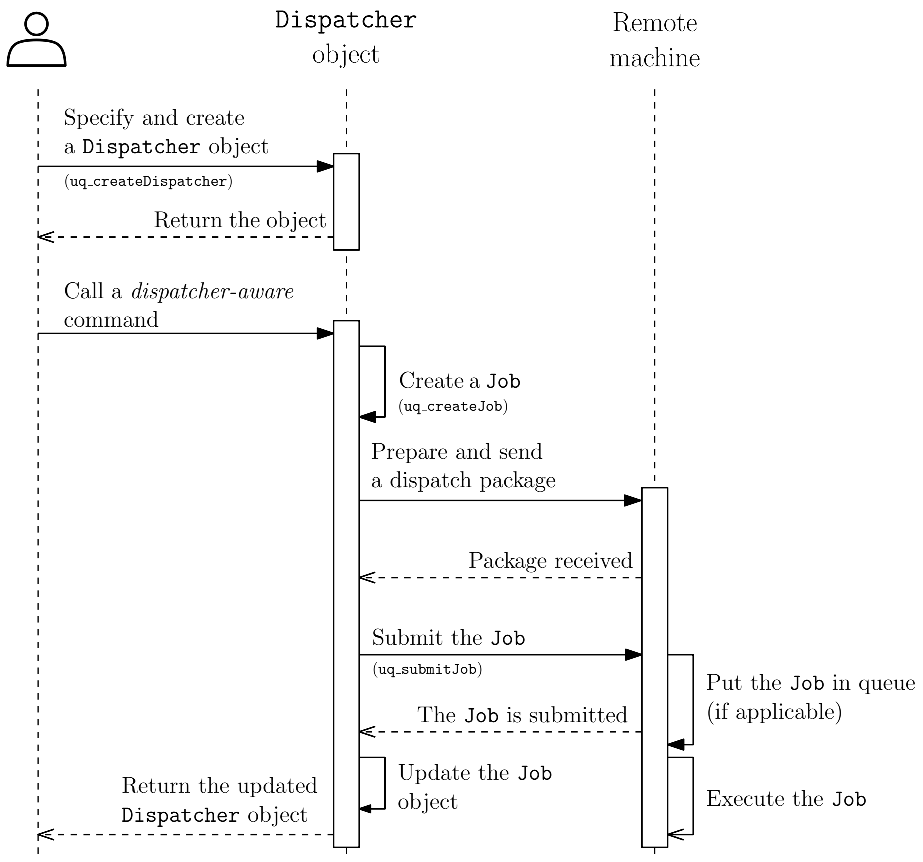


Figure 5: A sequence diagram depicting the process of dispatching a call to a dispatcher-aware command using a DISPATCHER object.

In practice, a JOB is an object that contains all the information relevant to the particular remote execution at hand; the information includes the (input) data, the operations that need to be executed (*e.g.*, function evaluations, metamodel training, UQLAB analysis), the required files, the number of CPUs used, etc. Each time users call a dispatcher-aware command, a JOB object associated with the remote execution of that function call is created by the DISPATCHER object.

During a given remote execution, a JOB progresses through a sequence of different stages. These different stages are represented by the `Status` property of the JOB; there are six of them:

- `'pending'`: A JOB has been created and the dispatch package has been sent to the remote machine ready for submission.
- `'submitted'`: When a JOB is submitted to a remote machine with a job scheduler, it enters the queueing system of the scheduler; it will remain as `'submitted'` until it reaches the top of the queue and the scheduler launches the JOB. If no scheduler is used (*e.g.*, for simple execution on a remote workstation), the JOB will immediately progresses to `'running'`.

- `'running'`: When a `'submitted'` JOB reaches the top of the queue and the scheduler (if used) launches the JOB, it progresses to `'running'` status.
- `'complete'`: If the remote execution has been completed successfully without errors, the JOB progresses to `'complete'` status.
- `'canceled'`: Users may deliberately cancel a `'submitted'` or `'running'` JOB to stop its execution.
- `'failed'`: If the remote execution fails or exits with errors, the JOB will have a `'failed'` as its status.

Note: The last three status (`'complete'`, `'canceled'`, and `'failed'`) will be collectively referred to as *finished* in the following.

A JOB is automatically created by a dispatcher-aware command. Once created, however, users may interact with JOB objects through several UQLAB helper functions. For instance, users can always query the status of a JOB. In more advanced usage, users have access to initiate a certain progression from one stage to another and, therefore, change the `Status` of a JOB. The progression are graphically shown in Figure 6.

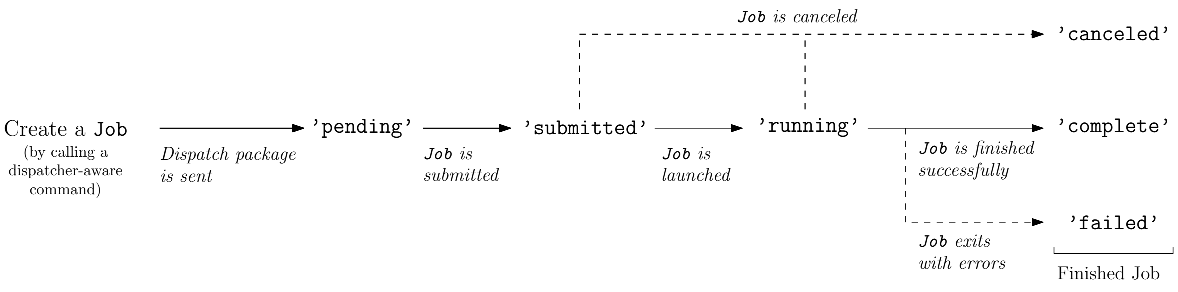


Figure 6: A JOB and the progression of its status.

Note: Due to the asynchronous nature of the remote computations, users may dispatch a new computation before the previous one is finished. From the user's perspective, this means a dispatcher-aware command can be called one after the other even before any of the executions in the remote machine from the previous calls are concluded.

1.3.4 Synchronized vs. non-synchronized execution

As introduced in Section 1.2, a dispatched computation is always executed asynchronously on the remote machine. The local UQLAB session and the execution on the remote machine are, in principle, independent of each other. However, the local UQLAB session and the remote execution are synchronized by default.

In the synchronized execution mode, the MATLAB command prompt is put on hold until the JOB reaches one of the available *finished* states (Figure 7). This results in a user experience that is very similar to the local execution, in which the next command can only be executed when the current one is finished.

Note: This is probably the preferred mode of execution when, for instance, a remote workstation licensed to run a simulation code is used to perform model evaluations, but the rest of the analysis should be performed instead locally.

Nevertheless, users may override this default by allowing for a non-synchronized execution if they wish. In the non-synchronized mode of execution, control is given back to the local UQLab session (and the Matlab command prompt). This happens regardless of the status of the remote computation, and whether the results are already available. The remote machine is allowed to execute and finish the Job on its own term.

Note: Because the results will not be available right after the Job has been submitted, in the non-synchronized execution mode, users must manually retrieve the results back once they become available. Section 1.3.4 explains this procedure further.

Figure 7 illustrates both modes of execution. In Chapter 2, examples of these two modes of executions are provided.

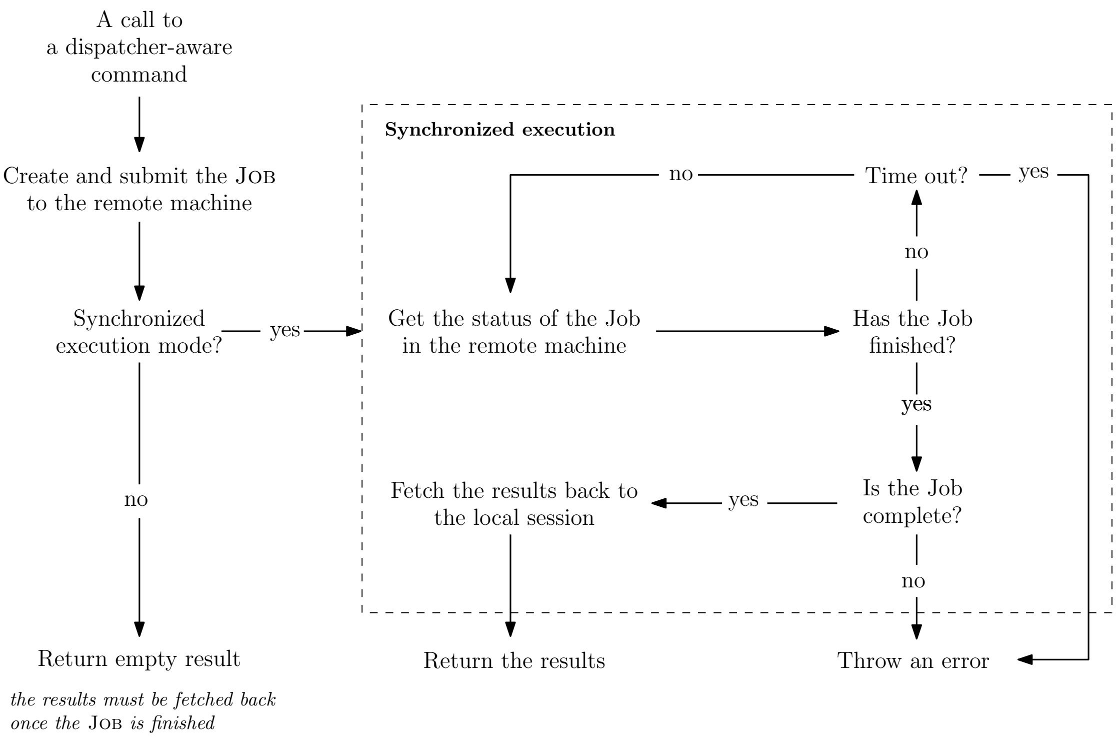

A call to a dispatcher-aware command
Create and submit the Job to the remote machine
Synchronized execution mode?
yes
no
Return empty result
the results must be fetched back once the Job is finished
Synchronized execution
Get the status of the Job in the remote machine
Has the Job finished?
no
Time out?
no
yes
yes
Is the Job complete?
yes
Fetch the results back to the local session
no
Return the results
Throw an error

Figure 7: Synchronized vs. non-synchronized execution mode. In the synchronized execution mode, the Dispatcher module waits for the remote execution to finish (or a timeout has been reached), fetches the results back to the local client, and returns the results to the local UQLab session before giving the control back to the user.

1.3.5 Fetching the results of a remote computation

When a computation is dispatched using the non-synchronized execution mode, control is given back to the local UQLAB session (and MATLAB command prompt). This happens regardless of whether the results are already available or not.

In order to access the results, users must explicitly request the results to be brought (*i.e.*, *fetched*) back to the local UQLAB session once the JOB reaches `'complete'` status. The process is illustrated in Figure 8.

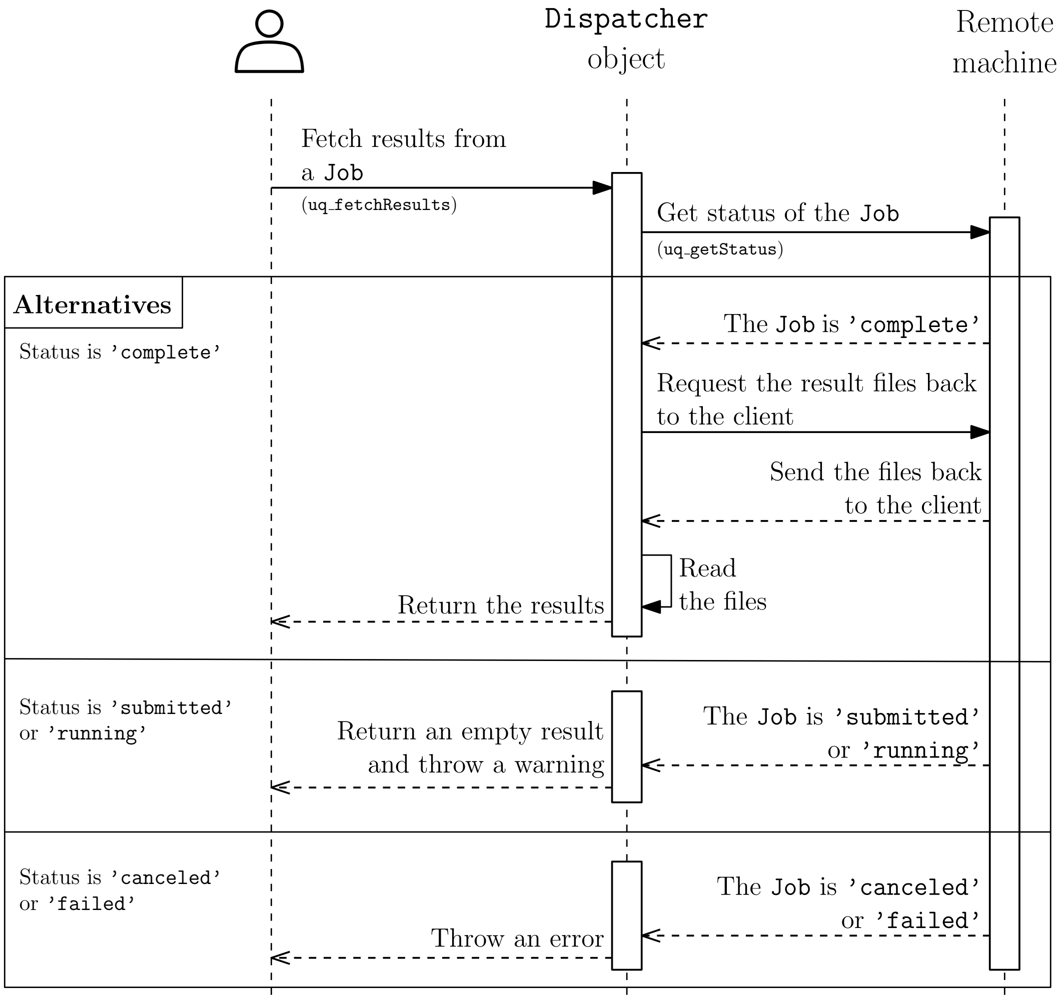

Dispatcher object
Remote machine
Fetch results from a Job
(uq_fetchResults)
Get status of the Job
(uq_getStatus)
Alternatives
Status is 'complete'
The Job is 'complete'
Request the result files back to the client
Send the files back to the client
Read the files
Return the results
Status is 'submitted' or 'running'
Return an empty result and throw a warning
The Job is 'submitted' or 'running'
Status is 'canceled' or 'failed'
The Job is 'canceled' or 'failed'
Throw an error

Figure 8: A sequence diagram depicting the process of fetching the results of remote computation back to the local UQLAB session.

By default, all the results from a JOB execution are kept in the remote machine. As long as the files remain in the remote machine, the results can always be fetched back by the DISPATCHER object regardles of the execution mode of the dispatched computation. If the local UQLAB session is closed, the associated JOBs may be retrieved for the new object in a new UQLAB session. Section 2.3.5.2 provides an example of this particular use case.

1.3.6 Parallel execution

Certain dispatched computations can be executed in parallel. The details of the parallelization depend on the particular implementation of dispatcher-aware commands. All the currently available dispatcher-aware commands are parallelizable. Specifically, the tasks carried out both by `uq_evalModel` and `uq_map` are of the *perfectly parralel* type. In this particular type of parallel calculations, the main computational task can be divided into subtasks that are completely independent of each other.

Both `uq_evalModel` and `uq_map` operate on a sequence of inputs. For instance, `uq_evalModel` evaluates a UQLAB MODEL on each row of the input matrix. In this case, each subtask consists of the same model evaluation but on a different subset of the whole input matrix.

Users may specify multiple remote processes to carry out the computation. In this case, the DISPATCHER module can automatically split the input data of a dispatched computation into slices and submits the computation on each slice in parallel to the remote machine. When the results are fetched, DISPATCHER automatically merges the results from each slice before returning them to the local UQLAB session in the proper format. In other words, the parallel execution in the remote machine is transparent for users. Figure 9 illustrates this particular workflow.

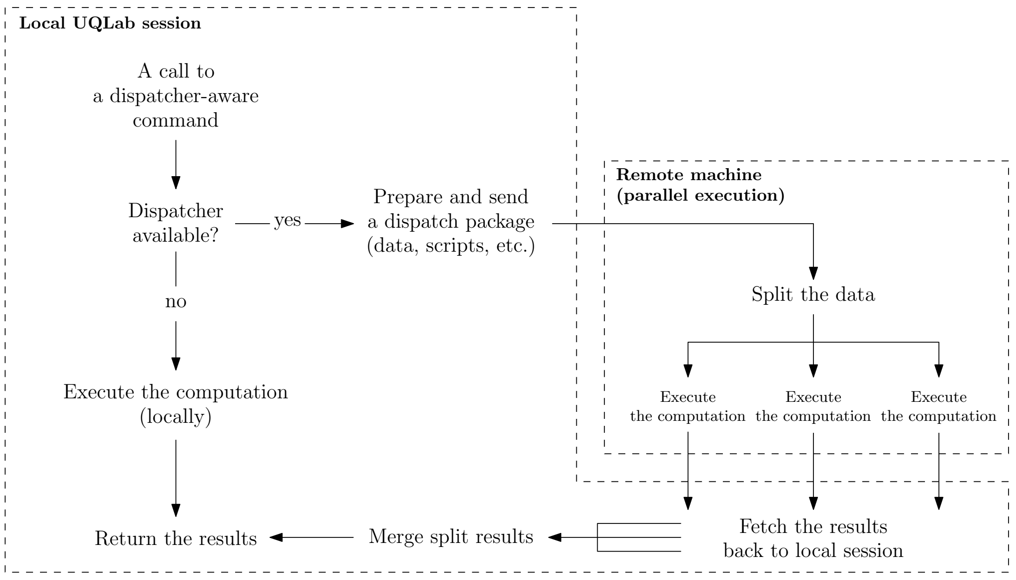

Local UQLab session
A call to a dispatcher-aware command
Dispatcher available?
yes
no
Prepare and send a dispatch package (data, scripts, etc.)
Execute the computation (locally)
Return the results
Merge split results
Remote machine (parallel execution)
Split the data
Execute the computation
Execute the computation
Execute the computation
Fetch the results back to local session

Figure 9: A simplified view of the workflow for a perfectly parallel computing task carried out by `uq_evalModel` and `uq_map`.

Note that the perfectly parallel problem is just one type of parallel computing problem. Depending on other dispatcher-aware commands that may become available in UQLAB in the future, other—more complex—types of parallel computing problem[4] may also be tackled using the DISPATCHER module.

[4]in which the subtasks are not completely independent; that is, there may be communications (*e.g.*, data transfers) between them.

Chapter 2

Usage

This chapter presents usage examples of the HPC DISPATCHER module based on the two available dispatcher-aware commands (`uq_evalModel` and `uq_map`, see Section 1.3.1). First, a reference problem is set up to showcase how a MODEL evaluation (*i.e.*, by using `uq_evalModel`) can be dispatched to a remote machine. The subsequent sections then present the essential steps of dispatching the evaluation, from setting up a DISPATCHER object to fetching the results back to the local UQLAB session.

Section 2.4 deals with a more advanced problem of a parametric study to evaluate the performance of a wide class of metamodels. This problem showcases how `uq_map` can be dispatched to a remote machine to solve a computationally expensive parametric study. Table 1 lists the various terminologies that are relevant to the discussion in this chapter.

2.1 Reference problem: dispatching the evaluation of a UQLAB MODEL

To showcase a dispatched MODEL evaluation, a UQLAB MODEL will be created based on the Ishigami function (Ishigami and Homma, 1990), as shown in Chapter 2 of UQLAB User Manual – the MODEL module. The Ishigami function is an analytical 3-dimensional function given by the following equation:

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

where the parameters a and b are set to 7 and 0.07, respectively. The input domain of the function is $[-\pi, \pi]$ for each input variable.

The vectorized implementation of the function is available in UQLAB as `'uq_ishigami.m'` and it reads:

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

Table 1: List of terminologies in alphabetical order relevant to the discussion in this chapter.

Term	Description
Dispatched computation	A UQLAB computation executed on the remote machine. The relevant information associated with a dispatched computation is stored by the DISPATCHER module in a JOB object.
Job	A computational task executed on the remote machine; often managed by a scheduler.
Key-based authentication	An authentication method based on the exchange of an encrypted SSH *key-pair* for secure communication between the local client and the remote machine.
Local client	A personal computer (PC, *e.g.*, laptop or desktop) running a UQLAB session from which users dispatch their computation.
Local UQLAB session	A UQLAB session running on the local client.
Remote machine	Computing resources that execute the dispatched computations; may be located anywhere in the world. Users typically do not have physical access to interact with them (therefore, *remote*). A remote machine may refer to a single server, an HPC cluster, or a cloud server.
Remote machine profile file	A MATLAB script that stores the information about a remote computing environment; it is used to set up the DISPATCHER module so that computations can be dispatched to the remote machine.
Remote MATLAB/UQLAB session	A MATLAB (and UQLAB) session running on the remote machine.
Scheduler	A program to manage the computational tasks (*i.e.*, jobs) submitted to computing resources (*e.g.*, HPC clusters). Scheduler manages job queues and ensures that each job is executed with the resources it needs.
SSH	*Secure Shell*, an encrypted network protocol; it used to securely log in from one computer to another via the network

A MODEL object using this function can then be created in UQLAB as follows:

```
% Start the UQLab framework (if not already started)
uqlab

% Define the MODEL options
ModelOpts.mFile = 'uq_ishigami'

% Create the MODEL object
myModel = uq_createModel(ModelOpts);
```

2.2 Setting up and creating a DISPATCHER object

To dispatch a MODEL evaluation to a remote machine using `uq_evalModel`, a DISPATCHER object must first be created. This section presents how to set up and then create such an object.

2.2.1 Requirements

Before creating a DISPATCHER object, users need to make sure a few requirements outside the UQLAB environment are met. Specifically, they need to:

- have an account and access (via SSH) to a remote machine, including a remote folder with read/write access permissions;
- set up a secure key-based authentication for the SSH connection to the remote machine (see Appendix A for detailed step-by-step instructions);
- set up a profile file of the remote machine that stores the information about the remote environment and the tools therein (see Appendix B for detailed step-by-step instructions and the available templates).

Furthermore, the local client (*i.e.*, the user's personal computer) requires a supported SSH client: PuTTY for Windows machines or OpenSSH for Unix (Linux and macOS) machines. While OpenSSH is, by default, available on a Linux and MacOS operating systems, PuTTY must usually be installed separately on a Windows machine.

The following requirements hold for the remote machine/login node:

- **[Required]** It must run a Linux operating system. While the client (local) machine may be running Windows, macOS, or a Linux operating system, the remote OS is currently restricted to Linux. Additionally, it must have the `Bash` shell installed.

  Note: The following command line tools must also be available in the remote machine: `cat`, `chmod`, `command`, `date`, `printf`, `test`, and `type`.

- **[Required]** It must have an MPI implementation (Open MPI, MPICH, etc.) installed. Specifically, the command `mpirun` must be available.

Finally, depending on the actual usage of the DISPATCHER module, several additional tools may be required in the remote machine:

- **[Optional]** MATLAB ($\geq$ R2015b), if the user wishes to dispatch MATLAB computations remotely.

  Note: Note that without having MATLAB installed in the remote machine, the type of computation available to be dispatched is presently restricted to the evaluation of UQLink-based MODELS (see Section 2.3.5.5 for an example).

- **[Optional]** UQLAB with a valid license, if UQLab commands are dispatched, or its objects are needed in the remote machine. For instance, to dispatch the evaluation of UQLAB MODEL objects that execute MATLAB functions.

If UQLAB is used then MATLAB must also be available.

> **Note:** In the most common licensing schemes (e.g. academic and teaching licenses), the same UQLAB license can be used both on the client and the remote machine, as long as it is operated by the same user.

> **Note:** Before moving on to the next sections, make sure that an SSH key-based authentification to a remote machine and a remote machine profile file have been set up properly. Users are referred to Appendix A and B to set up the SSH connection and profile file, respectively. If you lack access to an available remote machine or need to learn more about the remote machine, consult the relevant documentation or contact your system administrator.

2.2.2 Creating a DISPATCHER object

Once the SSH key-based authentication to the remote machine and the profile file have been set up, a DISPATCHER can be created as follows:

```
DispatcherOpts.Profile = 'myHPCProfile';
myDispatcher = uq_createDispatcher(DispatcherOpts);
```

It is assumed that the profile file has been saved as `'myHPCProfile.m'` and it is available in the MATLAB search path.

> **Note:** UQLAB comes with a couple of remote profile template files to serve as a starting point. These template files are located in `$UQ_ROOTPATH\Profiles\HPC`. See Appendix B for detailed step-by-step instructions on how to prepare this file.

When a DISPATCHER is created, the SSH connection with a key-pair authentication is verified; an error is thrown if the connection cannot be established.

Some information of the resulting DISPATCHER object can be printed as follows:

```
uq_print(myDispatcher)
```

```
%----------------- Dispatcher object ---------------%
Object Name:             Dispatcher 1
Number of Processes:     1
Number of Jobs:          0

Remote Profile
   Name:                 myHPCProfile
   Username:             jdoubt
   Hostname:             cluster.mydomain.com
   Remote location:      /home/jdoubt/temp
%---------------------------------------------------%
```

By default, the number of remote processes used in a dispatched computation is set to 1. As shown in the `Number of Jobs` field, there is currently no dispatched computation associated with the DISPATCHER object.

2.3 Dispatching MODEL evaluations

Once the DISPATCHER has been created, a MODEL evaluation can be dispatched to the remote machine by adding the `'HPC'` flag to the `uq_evalModel` command. For instance, to evaluate remotely the Ishigami function previously created on point $\mathbf{x} = [1, 1, 1]$, the following command is used:

```
Y = uq_evalModel([1 1 1], 'HPC')
```

By default, once the computation is dispatched, the DISPATCHER waits for the remote execution to finish before returning the results (see Section 1.3.4 on synchronization):

```
Checking the status of the remote execution...
Checking the status of the remote execution...
Job Status: 'complete' reached.

Y =

    5.8821
```

Notice that other than the messages regarding the checking the status of the remote execution, this model evaluation works exactly the same as the local execution:

```
Y = uq_evalModel([1 1 1])

Y =

    5.8821
```

Note: Users may change the display setting of the DISPATCHER object to allow for more or less information to be displayed when a computation is dispatched. See the `Display` option in Table 7 for details.

2.3.1 Keeping track of the dispatched evaluation

The DISPATCHER object created in Section 2.2.2 can now be used to track and modify the status of the JOB, if needed. Printing the DISPATCHER object now provides additional information about the JOB associated with the dispatched computation. For example:

```
uq_print(myDispatcher)
```

may give information similar to:

```
%----------------- Dispatcher object -----------------%
```

```
    Object Name:              Dispatcher 1
    Number of Processes:      1
    Number of Jobs:           1

    Remote Profile
       Name:                  myHPCProfile
       Username:              jdoubt
       Hostname:              cluster.mydomain.com
       Remote location:       /home/jdoubt/temp

    Job Details (1)
       Status:                complete
       JobID:                 1768
       Execution mode:        Synchronized ('sync')
       Num. of out. args.:    1
       Remote folder:         /home/jdoubt/temp/12Dec2020_at_10442307
       Submit:                12/21/20 10:44:23 AM (UTC)
       Start:                 12/21/20 10:44:23 AM (UTC)
       Finish:                12/21/20 10:44:30 AM (UTC)
       Last Update:           12/21/20 10:44:41 AM (UTC)
       Queue duration:        00 hrs 00 mins 00 secs
       Running duration:      00 hrs 00 mins 07 secs
%---------------------------------------------------%
```

The information showed indicates that there is one JOB associated with the DISPATCHER object, and the JOB has finished its execution successfully (*i.e.*, having a `'complete'` status).

Note: For consistency reasons, when using remote machines located all around the world, the date and time information, as shown from printing out a DISPATCHER object, are always given in UTC (Universal Time Coordinated), the primary time standard.

The current status of the JOB can be queried by typing:

```
uq_getStatus(myDispatcher)
```

which, for the current dispatched computation, returns:

```
ans =

  'complete'
```

2.3.2 Execution mode of the dispatched computation

As exemplified above, by default, the execution of a dispatched computation is carried out in the synchronized execution mode. This means once the JOB is submitted to the remote machine, the DISPATCHER module waits for the execution to finish, and, once finished, fetch the results from the remote machine back to to the local UQLAB session. In other words, the remote execution is *synchronized* with the local UQLAB session.

Users may choose to carry out the dispatched computation in the non-synchronized mode (*i.e.*, asynchronously). To dispatch a computation in this mode, make sure that the property `ExecMode` of the DISPATCHER object is set as follows:

```
myDispatcher.ExecMode = 'async';
```

Note: By default, the value of `ExecMode` property is set to `'sync'`.

In the non-synchronized execution mode, once the JOB is submitted to the remote machine, the control is immediately given back to the user. The remote machine can start, carry out, and finish the computation on its own term. For example, once the property `ExecMode` is changed to `'async'`, the same evaluation can be dispatched:

```
Y = uq_evalModel([1 1 1],'HPC')
```

```
Y =

   []
```

By default, if the dispatched computation is not synchronized and the results of the computation are assigned to a variable, the DISPATCHER module returns an empty vector.

As before, the current status of the JOB can be queried by typing:

```
uq_getStatus(myDispatcher)
```

which, if the remote execution is still ongoing, will return:

```
ans =

   'running'
```

Note: Depending on the status of the remote execution, the command may return other status; see Section 1.3.3 for details on the status of remote execution.

If the JOB has not finished, the following command can be used to lock the MATLAB execution until the JOB is finished:

```
uq_waitForJob(myDispatcher)
```

```
Checking the status of the remote execution...
Checking the status of the remote execution...
Job Status: 'complete' reached.
```

This command is useful, for instance, to pause the execution until the results of the dispatched computation are available. For additional details, see Section 1.3.4.

2.3.3 Fetching the results of the dispatched computation

Once a JOB has been *successfully* completed (*i.e.*, it reaches `'complete'` status), the computation results need to be fetched back to the local UQLAB session. The results of a dispatched computation are, by default, kept in the remote machine. The command `uq_fetchResults` fetches the result back to the local session. For example:

```
Y = uq_fetchResults(myDispatcher)
```

```
Y =

    5.8821
```

The results of a dispatched computation are exactly the same as the ones that would be produced by calling `uq_evalModel` without the `'HPC'` flag (*i.e.*, executed locally):

```
Y = uq_evalModel([1 1 1])
```

```
Y =

  5.8821
```

The `uq_fetchResults` command works for any completed JOBs regardless of their execution mode (*i.e.*, `'sync'` vs. `'async'`). By default, the results of dispatched computations are kept on the remote machine. As long as the remote directory in which the results are stored remains intact, the results can be fetched multiple times using the same DISPATCHER object[1].

Note that only the results of complete JOBs can be fetched back. If a JOB still has the status `'submitted'` or `'running'`, an attempt to fetch the results back will return an empty array accompanied by a warning message. For instance, provided that the JOB is still running, fetching the results will gives the following:

```
Y = uq_fetchResults(myDispatcher)
```

```
Warning: Can't fetch results; Job is running!
(Type "warning off " to suppress this warning.)

> In uq_fetchResults_uq_default_dispatcher (line 49)
  In uq_fetchResults (line 69)

Y =

    []
```

Finally, if a JOB fails for any reason or has already been canceled, an attempt to fetch the results will throw an error.

[1]or a DISPATCHER object having the same setting; see Section 2.3.5.2.

2.3.4 Parallel execution

When there are multiple model evaluations to execute, and the remote machine has more than one computational core available, the MODEL evaluation can be executed in parallel. For instance, to create two parallel remote processes, for the evaluation[2] modify the DISPATCHER object property `NumProcs` as follows (see Section 1.3.6 for details):

```
myDispatcher.NumProcs = 2;
```

Dispatch a model evaluation on multiple sets of input parameters as follows:

```
X = [1 1 1; 0 0 0; 0.5 0.5 0.5; -1 -1 -1];
uq_evalModel(X,'HPC')
```

Because there are four points in the input and two parallel remote processes, each process will carry out the evaluation of two input points.

After waiting the JOB to finish:

```
uq_waitForJob(myDispatcher)
```

```
Checking the status of the remote execution...
Job Status: 'complete' reached.
```

the results of the dispatched computation can be fetched:

```
Y = uq_fetchResults(myDispatcher)
```

```
Y =

    5.8821
         0
    2.0914
    4.0309
```

The parallel execution on the remote machine is transparent to users. As it can be seen, the end results are the same whether the MODEL evaluation is carried out with a single process or multiple processes.

Once again, the results are identical to the results of local computation (*i.e.*, without the `'HPC'` flag):

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

```
Y =

    5.8821
         0
    2.0914
```

[2] `NumProcs` should be less or equal than the total number of the available computational cores in the remote machine.

```
    4.0309
```

2.3.5 Advanced usage

2.3.5.1 Multiple dispatched computations

Multiple computations can be dispatched one after the other. In the non-synchronized mode of execution, the remote executions of the previously dispatched computations need not be finished; they may not even be started on the remote machine (*i.e.*, still in the queue of the remote scheduler).

After making sure that the execution mode is non-synchronized:

```
myDispatcher.ExecMode = 'async';
```

multiple calls to `uq_evalModel`:

```
uq_evalModel([1 2 3],'HPC')
uq_evalModel([-1 0 1],'HPC')
uq_evalModel([0 0 -1],'HPC')
```

will create multiple dispatched computations (*i.e.*, JOBS).

To see an overview of all the JOBS associated with a DISPATCHER object, use the `uq_listJobs` command on the DISPATCHER object:

```
uq_listJobs(myDispatcher)
```

which may results in the following:

```

Dispatcher Object: Dispatcher 2

No.  Job ID  Status      Tag                                                        Finish ...
                                                                                        ...
 1   1871    complete    uq_evalModel of <Model 1> on <08-Mar-1978 14:02:34>   03/08...
 2   1872    complete    uq_evalModel of <Model 1> on <31-Aug-1987 14:02:39>   08/31...
 3   1873    complete    uq_evalModel of <Model 1> on <11-May-1997 14:02:45>   10/15...
 4   2094    submitted   uq_evalModel of <Model 1> on <15-Jan-2001 14:18:02>
 5   2095    submitted   uq_evalModel of <Model 1> on <04-Jul-2012 14:18:08>
 6   2096    submitted   uq_evalModel of <Model 1> on <11-Apr-2063 11:00:00>
```

When there are multiple JOBS associated with the DISPATCHER object, all the helper commands shown so far work with a particular JOB by specifying its index as the second argument. For instance, to update and get the status of the JOB number 6, use:

```
uq_getStatus(myDispatcher,6)
```

The same syntax also applies for `uq_print`, `uq_waitForJob`, and `uq_fetchResults` previously shown. By default, when the JOB index is not specified, the command will operate on the last created JOB.

2.3.5.2 Saving and loading a DISPATCHER object

Because the execution of a dispatched computation can be detached from the local MATLAB session (*i.e.*, in the non-synchronized execution mode), the latter can be closed (and the computer shutdown) without interrupting the remote execution. However, to enable a later retrieval of the remote JOBs, all the information stored in the DISPATCHER object needs to be preserved in the local client by saving the object to a file. This can be easily achieved with `uq_saveDispatcher` command. For example, the following command:

```
uq_saveDispatcher('mySavedDispatcher',myDispatcher);
```

saves the DISPATCHER object `myDispatcher` to a file called `'mySavedDispatcher.mat'`. For a complete reference on the use of the command, refer to Section 3.5.

The command `uq_loadDispatcher` can be used in a new UQLAB session to restore the previously saved DISPATCHER object stored in a file. For example, the command:

```
myDispatcher = uq_loadDispatcher('mySavedSession');
```

loads the DISPATCHER object stored in `'mySavedSession.mat'` and returns the object as `myDispatcher`.

Note: By default, loading a DISPATCHER object will also automatically add the object to the current UQLAB session. See Section 3.6 for details.

Note: If `uq_loadDispatcher` is called without a filename argument, then, by default, it looks in the current working directory for a MAT-file created automatically when a DISPATCHER object is created.

If the remote folder specified in the DISPATCHER object is still intact[3], all the JOBs associated with the DISPATCHER object can be queried, manipulated, and their results fetched exactly as if they were computed locally, as described in the beginning of Section 2.3.

Alternatively, if the DISPATCHER object has not been saved before the current MATLAB session is closed, JOBs stored in a remote directory may still be recovered using a newly created DISPATCHER object (thus has no JOBs associated with it) with the same profile file.

For instance, by creating a new DISPATCHER object with an identical configuration options as before (*i.e.*, mainly by using the same remote profile file):

```
myNewDispatcher = uq_createDispatcher(DispatcherOpts);
```

the remote JOBs can be retrieved by calling the following command:

```
uq_retrieveJobs(myNewDispatcher)
```

[3] in which all the files related to the remote executions are stored

The command will search through the remote location specified in the DISPATCHER object and retrieve any JOB objects within. The DISPATCHER object will be updated with the retrieved JOBs. The command may also be called with a non-empty DISPATCHER object. In this case, an identical JOB with what is already in the DISPATCHER object will not be added.

Note: The retrieving process looks for the sub-directories (one level) within the specified remote folder. By default, a JOB-specific remote directory name is based on the timestamp of its creation and the retrieved JOBs will be ordered chronologically.

2.3.5.3 MODELS with multiple output arguments

UQLAB MODELS may return more than one output at a time ($N_{\text{out}} > 1$). Some MODELS may be set up to return a $N \times N_{\text{out}}$ matrix, while others may return multiple output arguments[4]. Both of these forms are fully supported in a dispatched computation.

If the MODEL returns a matrix, then the dispatched computation will also automatically returns a matrix. If the MODEL uses multiple output arguments, then the evaluation can be called with multiple output arguments when the model evaluation is dispatched. For instance, if a MODEL supports two output arguments, then:

```
[Y1,Y2] = uq_evalModel(X,'HPC')
```

requests two outputs from the MODEL evaluation.

Multiple output arguments are also supported when the results are fetched as long as the computation is dispatched with the same number of output arguments. For instance, the following command fetches the results of a dispatched computation having two outputs:

```
[Y1,Y2] = uq_fetchResults(myDispatcher)
```

Note: The ability to fetches multiple outputs from a dispatched MODEL evaluation depends on the underlying MODEL itself and how the evaluation is dispatched initially. For instance, if a MODEL evaluation is dispatched with only a single output arguments, then the results cannot be fetched for more than one output, regardless of whether the underlying MODEL supports multiple output arguments or not.

2.3.5.4 Attaching m-files for m-file-based MODELS

As previously explained in Section 1.3, the DISPATCHER object is responsible for preparing and sending a dispatch package to the remote machine. This package includes relevant data and scripts required for the remote execution. When a MODEL object is created based on a

[4]such as in the case of a UQLAB Kriging metamodel, in which the first and second output is the Kriging mean and variance, respectively.

function inside a local `m`-file, this dispatch package automatically includes the `m`-file.

However, if the function file depends on one or more additional user-defined files, the DISPATCHER object will not be able to automatically trace these files and include them on the dispatch package. The solution to this problem is to manually copy these files to the remote machine and make them available to the remote MATLAB session via the `AddToPath` property of the DISPATCHER object.

For instance, suppose the required files are already available in `'/home/jdoubt/matlibs'` of the remote machine, the `AddToPath` property can be modified as follows:

```
myDispatcher.AddToPath = '/home/jdoubt/matlibs';
```

During the remote execution, the specified directory will be added to the MATLAB search path and, therefore, all the files within will be visible to the remote MATLAB session. Multiple remote directories can be added using a cell array that contains the different different directories. For instance, the following specification:

```
myDispatcher.AddToPath = {'/common/libs', '/home/jdoubt/matlibs',...
  '/home/jdoubt/data'};
```

adds three different directories to the remote MATLAB search path.

Finally, if the additional required files are scattered across multiple sub-directories of the same parent directory, then `AddTreeToPath` property may be used instead. In contrast with the `AddToPath` property, any directories listed in `AddTreeToPath` will be added to the remote MATLAB path, along with all their sub-directories.

As an example, to make all the following files:

```
/home/jdoubt/myCode
 |__PreProcessor.m
 |__solver
 |    |__SolverStepA.m
 |    |__SolverStepB.m
 |__postprocessor
      |__PostProcessorStepA.m
      |__PostProcessorStepB.m
```

available in the remote MATLAB search path, it is suffice to specify `AddTreeToPath` property as follows:

```
myDispatcher.AddTreeToPath = '/home/jdoubt/myCode';
```

All the sub-directories within `'/home/jdoubt/myCode'`, *i.e.*, `'solver'` and `'postprocessor'` will be automatically added to the remote MATLAB search path. Note that if this directory was instead specified in `AddToPath`, only `'PreProcessor.m'` would be visible to the remote MATLAB session.

Similar to AddToPath, multiple parent directories can be specified to AddTreeToPath using a cell array.

Note: The values of AddToPath and AddTreeToPath properties apply to all JOBs created using the DISPATCHER object.

2.3.5.5 Dispatching a UQLINK-based MODEL evaluation

While dispatching UQLINK MODEL evaluations using uq_evalModel has the same workflow as the other UQLab models, users need to be aware of several particularities. UQLINK-based models slightly differ from other UQLab models due to their dependency on third-party simulation codes. At the same time, evaluations of UQLINK models based on third-party codes may not require MATLAB (and, therefore, UQLAB) at all. If this is the case, the remote machine to which model evaluations are dispatched does not need to have MATLAB nor UQLAB installed.

The example below is based on a Python implementation of the Ishigami function. It is assumed that the reader is familiar with UQLINK MODELs and their options; while the important steps are repeated for completeness, detailed explanations for all the options of UQLINK can be found in Moustapha et al. (2021).

Problem setup

A Python3 implementation of the Ishigami function (Eq. (2.1)) is given as follows:

```
#!/usr/bin/env python3

import sys
import numpy as np

def ishigami(X):
    Y = np.sin(X[:,0]) + 7*np.sin(X[:,1])**2 +
      0.1*(X[:,2]**4) @ np.sin(X[:,0])
    return Y


def main():
    X = np.loadtxt(sys.argv[-2], delimiter=',', ndmin=2)
    Y = ishigami(X)
    np.savetxt(sys.argv[-1], Y, delimiter=',')


if __name__ == '__main__':
    main()
```

The script is saved as uq_ishigami.py. The script takes a simple comma-separated-values (CSV) file as the input and produces the output in another file. It can be executed from the command line as follows:

```
uq_ishigami.py inp.csv out.csv
```

Finally, it is assumed that this script has been made available on the remote machine in the `~/scripts` directory and that a Python3 interpreter and the `numpy` package are available on the remote machine.

The Python-based Ishigami function can be specified as a UQLINK MODEL as follows:

```
% Start the framework (if not already) started
uqlab
% Define the model options
ModelOpts.Type = 'UQLink';
ModelOpts.Command = 'uq_ishigami.py inp.csv out.txt';
ModelOpts.Template = 'inp.csv.tpl';
ModelOpts.Output.Parser = 'uq_read_ishigami';
ModelOpts.Output.FileName = 'out.csv';
ModelOpts.ExecutablePath = '~/scripts';
```

The template for the input is a csv file whose values have been replaced with keys (see UQLAB User Manual – The UQLINK module for details). It contains a single line as follows:

```
<X0001>,<X0002>,<X0003>
```

The output parser is a simple MATLAB function that reads:

```
function Y = uq_read_ishigami(outputfile)

Y = dlmread(outputfile);

end
```

The most important differences between local and dispatched MODEL evaluations are the meaning of the following two configuration options:

- `ExecutablePath`: In the dispatched MODEL evaluations, this path refers to the directory where the executable command is located *on the remote machine*. If the path is left empty, then it is assumed that the executable (as specified in the `Command` configuration option) belongs in the `PATH` environment variable of the remote machine.

  Note: Users are responsible to set up the executable of the third-party simulation code in the remote machine.

- `ExecutionPath`: In the dispatched model evaluations, this option becomes irrelevant (*i.e.*, any value will not be taken into account) as this path will be set up automatically by the DISPATCHER object, specific to a remote JOB to which the particular evaluation is attached.

The template file and the output parser must be available in the local MATLAB search path. Upon dispatching, they will be automatically included in the dispatch package. All the remaining options of UQLINK models have exactly the same meaning both for local and dispatched model evaluations.

The corresponding MODEL can be directly created without any further specification as follows:

```
myUQLinkModel = uq_createModel(ModelOpts);
```

To dispatch and evaluate the ishigami function on a single point:

```
Y = uq_evalModel([1 1 1],'HPC')
```

```
Checking the status of the remote execution...
Job Status: 'complete' reached.

Y =

    5.8821
```

As can be seen, other than the differences regarding the setup of the third-party code on the remote machine, dispatching a UQLINK MODEL evaluation and fetching back the results work the same as the other UQLAB models (see Sections 2.3.1-2.3.5).

The case with MATLAB/UQLAB installed on the remote machine

By default, dispatching `uq_evalModel` does not assume that both MATLAB and UQLAB are installed on the remote machine. The input files are created on the local client; the third-party code outputs are also parsed on the local client. If the size of these files is particularly large, copying these files between the local client and the remote machine via the network may take a long time[5].

In case MATLAB and UQLAB are installed on the remote machine, they may help to speed up the overall dispatched computation in some cases by avoiding the need to transfer (possibly) large files. Using both MATLAB and UQLAB on the remote machine, the input files and the output files are created and parsed on the remote machine. Only the numerical results of the dispatched computation are transferred back to the local client when the results are fetched.

To allow a dispatched UQLINK MODEL evaluation to use a remote MATLAB/UQLAB installation, use the following UQLINK configuration option (before the MODEL is created):

```
ModelOpts.RemoteMATLAB = true;  % The default is set to false
```

Current limitations

While most features and options of `uq_evalModel` on UQLINK MODELS work out of the box

[5]which would also depend on the network performance

when the evaluation is dispatched, note that some of the more advanced features of the UQLINK module are not supported by the DISPATCHER module. These are recovering failed results and resuming an analysis. See UQLAB User Manual – The UQLINK module (Sections 2.7.8 and 2.7.9) for details on these features.

2.4 Advanced use case: surrogate modeling parametric study with uq_map

This section showcases how `uq_map` (a UQLIB function) and the DISPATCHER module can be used to perform a distributed parametric study on surrogate models. The full computational model is the Ishigami function presented in Section 2.1 and the selected metamodel is Kriging (Lataniotis et al., 2021).

2.4.1 Problem statement

There are many options to select in the construction of a Kriging metamodel, from the correlation function to the trend function, and from the estimation method to the optimization method. Different choices of these options result in metamodels with varying predictive performance. For details on the Kriging metamodeling in UQLAB, see UQLAB User Manual – Kriging (Gaussian process modelling).

A parametric study is set up to compare the effects of using all possible combinations of a selected set of options. The options considered in the study are listed in Table 2.

Table 2: Considered options for the parametric study of Kriging metamodel construction. All of their combinations will be investigated.

Option	Value
Trend type	`'polynomial'`
Trend degree	0 (constant), 1 (linear), and 2 (quadratic)
Correlation family	`'exponential'`, `'gaussian'`, `'matern-3_2'`, and `'matern-5_2'`
Estimation method	`'CV'` (cross-validation) and `'ML'` (maximum-likelihood)
Optimization method	`'BFGS'` (Broyden-Fletcher–Goldfarb-Shanno), `'GA'` (genetic algorithm), and `'HGA'` (hybrid genetic algorithm)

The comparison of the different metamodels is based on a fixed-size experimental design with an independent validation set.

2.4.2 A quick introduction to uq_map

Before proceeding further with the parametric study use case, it is worthwhile to quickly introduce the UQLIB function `uq_map` (Section 1.3.1). `uq_map` may be executed locally or dispatched using the DISPATCHER module. The basic syntax of `uq_map` for local execution is as follows:

```
Y = uq_map(FUN,INPUTS)
```

where `FUN` (referred to as the *mapping* function) is a function handle and `INPUTS` is an input sequence. `uq_map` will evaluate `FUN` for each element of `INPUTS`. `INPUTS` may be a cell, structure, or numerical arrays (including, matrices). The output of `uq_map` is always a cell array whose dimension is consistent with `INPUTS`.

For example, `uq_map` can be used to evaluate the MATLAB built-in function `sum` on three different vectors of different length as follows:

```
inputs = {linspace(1,2,10),linspace(1,2,100),linspace(1,2,1000)};
Y = uq_map(sum,inputs)
```

```
Y =

  1x3 cell array

  {[15]}    {[150]}    {[1.5000e+03]}
```

A more comprehensive guide on the use of local `uq_map` can be found in the `uq_map` entry of the UQLIB user manual.

As a dispatcher-aware function, `uq_map` execution can be dispatched to a remote machine using a DISPATCHER object. The DISPATCHER object is specified as the third input argument with the following syntax:

```
Y = uq_map(FUN, INPUTS, DISPATCHEROBJ)
```

The simple example above can be dispatched using the DISPATCHER object `myDispatcher` as follows:

```
Y = uq_map(sum, inputs, myDispatcher)
```

```
Checking the status of the remote execution...
Job Status: 'complete' reached.

Y =

  1x3 cell array

  {[15]}    {[150]}    {[1.5000e+03]}
```

As can be seen, dispatching a `uq_map` execution is similar to dispatching a MODEL evaluation using `uq_evalModel`. The basic features of a dispatched computation explained before for `uq_evalModel` such as the two different execution modes (Section 2.3.2), getting the status of remote execution (Section 2.3.1), and fetching remote results (Section 2.3.3) work the same way for all dispatched `uq_map`.

Note: There are additional options that are specific to a dispatched `uq_map`. These options are provided in detail with examples in Appendix C.

2.4.3 Creating an experimental design and an independent validation set

To begin with the parametric study, an experimental design `Xtrain` and an independent validation set `Xval` common to all configurations are first created as follows:

```
% Specify the marginals:
for ii = 1:3
  InputOpts.Marginals(ii).Type = 'Uniform';
  InputOpts.Marginals(ii).Parameters = [-pi pi];
end

% Create an INPUT object:
myInput = uq_createInput(InputOpts);

% Generate training and validation samples:
rng(100,'twister')
Xtrain = uq_getSample(1e2);
Xval = uq_getSample(5e1);
```

Evaluate the MODEL on the experimental design and validation set:

```
Ytrain = uq_evalModel(Xtrain);
Yval = uq_evalModel(Xval);
```

2.4.4 Creating a parameter set

To create a set of configuration options, create a list for each of possible options as follows:

```
% Select Kriging as the metamodel Type:
ParametricOpts.Type = 'Metamodel';
ParametricOpts.MetaType = 'Kriging';

% Use polynomial trend with varying degree:
ParametricOpts.Trend.Type = 'polynomial';
ParametricOpts.Trend.Degree = {0, 1, 2};

% Use different correlation function options:
ParametricOpts.Corr.Family = {'exponential', 'gaussian',...
  'matern-3_2', 'matern-5_2'};

% Specify the methods for estimating the hyperparameters:
ParametricOpts.EstimMethod = {'CV','ML'};

% Use different optimization methods for hyperparameters calibration:
ParametricOpts.Optim.Method = {'BFGS', 'GA', 'HGA'};

% Use the fixed-sized training and validation set:
ParametricOpts.ExpDesign.X = Xtrain;
ParametricOpts.ExpDesign.Y = Ytrain;
ParametricOpts.ValidationSet.X = Xval;
ParametricOpts.ValidationSet.Y = Yval;
```

The fields and values of `ParametricOpts` are all valid options for a Kriging metamodel.

A set of configuration options can then be created as follows:

```
ParamSets = uq_createParameterSets(ParametricOpts);
```

The function `uq_createParameterSets` creates all the combination of the options by constructing a cartesian product of the listed option values. The construction results in 72 distinct configuration options to create Kriging metamodels.

2.4.5 Dispatching a parametric study using uq_map

Using `uq_map`, the function `uq_createModel` can be evaluated on each of the requested 72 configuration options created in the previous section. Coupled with a DISPATCHER object, the evaluation can be dispatched to a remote machine.

To dispatch the metamodel construction to the remote machine, use `uq_map` with the DISPATCHER object using `uq_createModel` as the mapping function and the requested configurations (`ParamSets`) as the input sequence as follows:

```
myKrigings = uq_map(@uq_createModel, ParamSets, myDispatcher,...
   'UQLab', true)
```

```
Checking the status of the remote execution...
Checking the status of the remote execution...
...
```

The named argument `'UQLab'` followed by a logical `true` is specified to load UQLAB in the remote computation.

Note: As explained in Section 2.2.1, UQLAB on the remote machine is only required if the remote computation requires UQLAB functionalities. In this case, metamodel construction is part of the UQLAB framework. Built-in MATLAB functions can be dispatched without the need for UQLAB in the remote machine.

Once the JOB has finished executing, the results are a cell array of all the Kriging metamodels created with the requested configurations:

```
myKrigings =

   72x1 cell array

   {1x1 uq_model}
   {1x1 uq_model}
   {1x1 uq_model}
   {1x1 uq_model}
```

```
  {1x1 uq_model}
  ...
```

Note: By default, any failure in a dispatched `uq_map` on the elements of the input sequence returns a `NaN`. To capture the error messages as well as more advanced error handling, see Appendix C.

2.4.6 Analyzing the results

Get the validation error from each Kriging MODEL object and convert the results to a simple vector:

```
valErrors = zeros(72,1);
for i = 1:72
  % Extract the validation error from each metamodel
  valErrors(i) = myKrigings{i}.Error.Val;
end
```

Sort the Kriging MODELS based on the validation errors in ascending order:

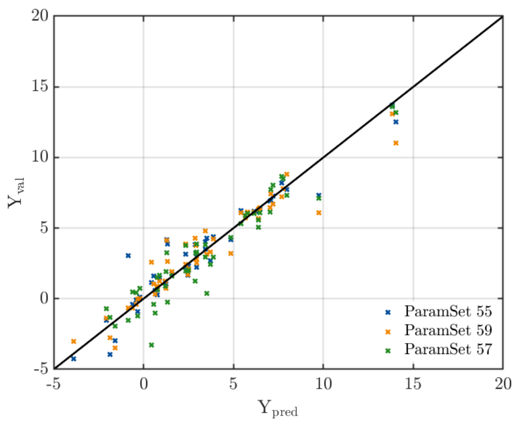
```
% Get the sorted indices of the validation errors
[~,sortedIdc] = sort(valErrors);
% Sort the Kriging metamodels
sortedKrigingModels = myKrigings(sortedIdc);
```

Finally, create a plot of predicted values vs. reference values in the validation set for the three best performing models:

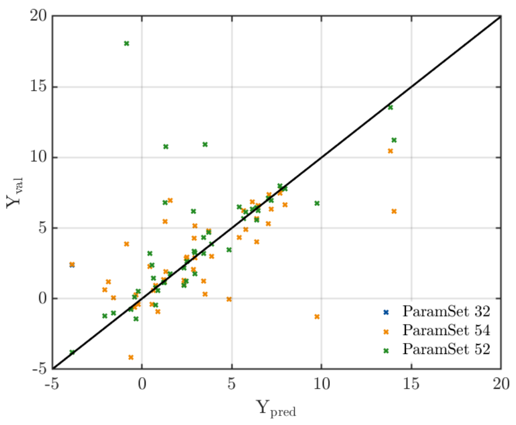
```
% Create an empty figure with the default UQLab formatting
uq_figure
uq_formatDefaultAxes(gca)
hold on
legendTxt = cell(3,1);

% Plot the metamodel predictions vs. validation data points
for i = 1:3
  uq_plot(Yval, uq_evalModel(sortedKrigingModels{i},Xval), 'x')
  legendTxt{i} = sortedKrigingModels{i}.Name;
end

% Create a 45-degree reference line
uq_plot([-5 15], [-5 15], 'k')
hold off

% Set an equal axis aspect ratio
xlim([-5 20])
ylim([-5 20])

% Add a legend
uq_legend(legendTxt, 'Location', 'southeast')

% Add axis labels
```

```
xlabel('$\mathrm{Y_{pred}}$', 'Interpreter', 'LaTeX')
ylabel('$\mathrm{Y_{val}}$', 'Interpreter', 'LaTeX')
```

The plot for the three worst performing models can be created in a similar fashion. The two plots are shown in Figure 10.



(a) The three best Kriging models



(b) The three worst Kriging models

Figure 10: The three top and worst Kriging models with respect to the validation error.

The actual options and the validation errors for the six selected models are summarized in Table 3.

Table 3: Considered options for the parametric study of Kriging metamodel construction.

Name	Trend degree	Corr. family	Estim. Method	Optim. Method	Validation error
ParamSet 55	2	'gaussian'	'CV'	'BFGS'	7.71×10^{-2}
ParamSet 59	2	'gaussian'	'ML'	'GA'	8.11×10^{-2}
ParamSet 57	1	'gaussian'	'CV'	'HGA'	8.76×10^{-2}
ParamSet 32	1	'exponential'	'ML'	'BFGS'	5.79×10^{1}
ParamSet 54	2	'exponential'	'ML'	'HGA'	5.79×10^{1}
ParamSet 52	1	'gaussian'	'CV'	'GA'	8.53×10^{1}

Note: This parametric study is simplified to illustrate the use of dispatched uq_map. It, by no means, follows best practices in surrogate modeling parametric studies.

Chapter 3

Reference List

How to read the reference list

Structures play an important role throughout the UQLab syntax. They offer a natural way to semantically group configuration options and output quantities. Due to the complexity of the algorithms implemented, it is not uncommon to employ nested structures to fine-tune the inputs and outputs. Throughout this reference guide, a table-based description of the configuration structures is adopted.

The simplest case is given when a field of the structure is a simple value or array of values:

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

which corresponds to the following syntax:

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

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

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

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

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

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

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

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

This is illustrated as follows:

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

Table Y: `Input.VALUE1`			
☐	`.Val1Opt1`	String	Description
☐	`.Val1Opt2`	Double	Description

Table Z: `Input.VALUE2`			
☐	`.Val2Opt1`	String	Description
☐	`.Val2Opt2`	Double	Description

Note: In the sequel, `double` and `doubles` mean a real number represented in double precision and a set of such real numbers, respectively.

3.1 Specify a remote machine profile file

A remote machine profile file is a MATLAB script that contains some variable definitions with information required by a DISPATCHER object to communicate with a remote machine. Table 4 provides the complete list of the variables.

Table 4: `Profile file variables`

●	`Hostname`	String	Hostname (or IP address) of the remote machine
●	`Username`	String	Username used to log in to the remote machine
●	`PrivateKey`	String	The private key file for a key-based authenticated SSH connection with the remote machine (see Appendix A for details).
⊕	`SavedSession`	String	The name of a PuTTY saved session. This option is only used if a saved PuTTY session is used to establish a passwordles SSH connection to the remote machine (see Appendix A for details).
●	`RemoteFolder`	String	The folder on the remote environment where users have read-write access. The dispatch package (data, scripts, etc.) is sent from the local client to the this folder.
☐	`Scheduler`	String default: `'none'`	The name of the scheduler used in the remote machine. Selecting a scheduler entails additional associated variables (*e.g.*, command to submit a job, option to set wall time). Several schedulers are supported by the HPC DISPATCHER module with default values for those variables.
		`'none'`	No scheduler is used in the remote machine
		`'slurm'`	SLURM workload manager
		`'lsf'`	Load Sharing Facility (LSF)
		`'pbs'`	Portable Batch System (PBS)
		`'torque'`	TORQUE resource manager
		`'custom'`	Custom scheduler; all the variables in `SchedulerVars` must be specified.

☐	SchedulerVars	Struct default: Table 3.1.1	Scheduler-specific variables. For non-'custom' schedulers, the default values for the associated variables are automatically set. If a custom scheduler is used or some of the associated variables need to be modified, users may specify them directly in the profile file.
☐	MATLABCommand	String default: ''	Fullpath to MATLAB executable on the remote environment. Note however, that in a shared computing resources, module system is in place and users may need to explicitly load MATLAB module on the remote environment before it can be used. See the note box below.
☐	MATLABOptions	String default: ''	Options used in the call to MATLAB on the remote machine.
☐	MATLABSingleThread	logical default: true	Flag to run remote MATLAB sessions using a single thread.
☐	RemoteUQLabPath	String default: ''	The name of the directory on the remote environment in which UQLab is located. This option is only used for dispatched computations that require UQLAB on the remote machine.
☐	EnvSetup	Cell array of strings default: {}	The list of commands to set up the remote environment; they are executed before a remote job is submitted. For instance, if an environment module system is in place (see the note box below), an MPI implementation may need to be explicitly loaded. If that is the case, this variable is the place to put the command.
☐	PrevCommands	Cell array of strings default: {}	The list of commands to run on each of the computing *nodes*. In a typical organization of an HPC cluster, a job is executed on nodes (*i.e.*, computing nodes) that are different from the node on which the job was submitted (*i.e.*, login node). For instance, any shared software across nodes (*e.g.*, MATLAB) must be loaded using commands specified in this variable. See the note box below.
☐	MPI	Struct default: Table 6	MPI implementation

Note: A common practice in a shared computing environment such as HPC clusters is to have an *environment modules system* in place (*e.g.*, Environment Modules, lmod for managing user's environment variables and toolsets. Consequently, while many common tools such as MPI or MATLAB may have been installed in the remote machine, they will not be available to the user without the user explicitly load these tools to their own environment. Consult the documentation or your system administrator, to learn more about the environment modules system of your remote computing environment.

3.1.1 Scheduler-specific variables

The following scheduler-specific variables are required for, among other things, a creation of a remote job script (a script the specify the resources requirement) and submitting and canceling the job.

Table 5: `SchedulerVars`

`.NodeNo`	String	Environment variable for the node number
	`'none'`	`'0'`
	`'slurm'`	`'$SLURM_NODEID'`
	`'lsf'`	`'0'`
	`'pbs'` / `'torque'`	`'$PBS_VNODENUM'`
`.WorkingDirectory`	String	Environment variable for the working directory of the submitted job
	`'none'`	`''`
	`'slurm'`	`'$SLURM_SUBMIT_DIR'`
	`'lsf'`	`'$LS_SUBCWD'`
	`'pbs'` / `'torque'`	`'$PBS_O_WORKDIR'`
`.HostFile`	String	Environment variable for the host file
	`'none'`	`''`
	`'slurm'`	`''`
	`'lsf'`	`''`
	`'pbs'` / `'torque'`	`'-hostfile $PBS_NODEFILE'`
`.Pragma`	String	Prefix in the directive in a job script
	`'none'`	`''`
	`'slurm'`	`'#SBATCH'`
	`'lsf'`	`'#BSUB'`
	`'pbs'` / `'torque'`	`'#PBS'`
`.JobNameOption`	String	Option to specify the job name

	'none'	''
	'slurm'	'--job-name=%s'
	'lsf'	'-J %s'
	'pbs' / 'torque'	'-N %s'
.StdOutFileOption	String	Option to specify a file into which the job standard output is redirected
	'none'	''
	'slurm'	'--output=%s'
	'lsf'	'-oo %s'
	'pbs' / 'torque'	'-o %s'
.StdErrFileOption	String	Option to specify a file into which the job standard error is redirected
	'none'	''
	'slurm'	'--error=%s'
	'lsf'	'-eo %s'
	'pbs' / 'torque'	'-e %s'
.WallTimeOption	String	Option to specify the job walltime requirement
	'none'	''
	'slurm'	'--time=%d'
	'lsf'	'-W %d'
	'pbs' / 'torque'	'-l walltime=00:%d:00'
.NodesOption	String	Option to specify nodes requirement
	'none'	''
	'slurm'	'--nodes=%d'
	'lsf'	''
	'pbs' / 'torque'	''
.CPUsOption	String	Option to specify CPUs requirement
	'none'	''
	'slurm'	'--ntasks-per-node=%d'
	'lsf'	'-n %d'
	'pbs' / 'torque'	''
.NodesCPUsOption	String	Option to specify both the nodes and CPUs requirement
	'none'	''
	'slurm'	''

	'lsf'	''
	'pbs' / 'torque'	'-l nodes=%d:ppn=%d'
.SubmitCommand	String	Scheduler command to submit a job
	'none'	''
	'slurm'	'sbatch'
	'lsf'	'bsub'
	'pbs' / 'torque'	'qsub'
.CancelCommand	String	Scheduler command to cancel a job
	'none'	'kill -15'
	'slurm'	'scancel'
	'lsf'	'bkill'
	'pbs' / 'torque'	'qdel'
.SubmitOutputPattern	String	Pattern to parse the job ID from the output of the submit command
	'none'	'[0-9]+'
	'slurm'	'[0-9]+'
	'lsf'	'(?<=<)[0-9]+(?=>)'
	'pbs' / 'torque'	'[0-9]+(.[a-zA-z0-9-_]+)+'

3.1.2 MPI-implementation-specific variables

Table 6: MPI

.Implementation	String default: 'OpenMPI'	Name of the MPI implementation. Supported implementation includes: 'OpenMPI', 'MPICH', 'MVAPICH', and 'IntelMPI'.
.RankNo	String	Environment variable for the MPI rank
	'OpenMPI'	'$OMPI_COMM_WORLD_RANK'
	'MPICH'	'$PMI_RANK'
	'MVAPICH'	'$PMI_RANK'
	'IntelMPI'	'$PMI_RANK'

3.2 Create a DISPATCHER object

Syntax

```
myDispatcher = uq_createDispatcher(DispatcherOpts)
```

Input

The structure variable `DispatcherOpts` contains the configuration information for a DISPATCHER object. The detailed list of available options is reported in Table 7.

Table 7: `DispatcherOpts`

●	`.Profile`	String	Name or filename of the profile file to be used. The profile file must be in the MATLAB search path. See Appendix B for details.
☐	`.Type`	String default: `'uq_default_dispatcher'`	Select the DISPATCHER type
☐	`.Name`	String	Unique identifier for the DISPATCHER object
☐	`.Display`	String default: `'standard'`	Level of information displayed during dispatched computations
		`'quiet'`	Minimum display level, displays nothing or very few information.
		`'standard'`	Default display level, shows the most important information.
		`'verbose'`	Maximum display level, shows all the information on runtime, like updates on iterations, etc.
☐	`.LocalStagingLocation`	String default: `''`	Local directory in which temporary files of the dispatch package is created before sent to the remote
☐	`.NumProcs`	Integer scalar default: 1	Number of parallel processes used on the remote machine
☐	`.NumCPUs`	Integer scalar default: 1	Number of requested CPUs on the remote machine. If this number is smaller than `.NumProcs`, the DISPATCHER module automatically requests the same number of CPUs for dispatched computations.

⊞	`.NumNodes`	Integer scalar default: 1	Number of nodes used to distribute the computation. It is assumed that all nodes have the same number of CPUs and that all of them together sum up to `.NumCPUs`.
⊞	`.CPUsPerNode`	Integer scalar	Number of CPUs available on each node. The number of nodes is computed automatically from this value. If both this option and `.NumNodes` are given, this option is ignored.
☐	`.AddToPath`	String or cell default: `{}`	List of paths on the remote machine to be added to the PATH of the remote execution environment (*e.g.*, MATLAB search path).
☐	`.AddTreeToPath`	String or cell default: `{}`	List of paths on the remote machine to be added to the PATH of the remote execution environment (*e.g.*, MATLAB search path). All the subdirectories of the listed paths will be added to the PATH.
☐	`.ExecMode`	String default: `'sync'`	Execution mode of the dispatched computation (see Section 1.3.4 and Section 2.3 for details).
		`'sync'`	Synchronized execution mode
		`'async'`	Non-synchronized (*asynchronous*) execution mode
☐	`.SyncTimeOut`	Double scalar default: `Inf`	Elapsed time (in seconds) before an attempt to carry out synchronous mode of execution fails
☐	`.CheckInterval`	Double scalar default: 5	Elapsed time (in seconds) before another attempt to update the status of a JOB is made
☐	`.JobWallTime`	Double scalar default: 60	Runtime limit (in minutes) for the remote execution, if a job scheduler is used
☐	`.FetchStreams`	Logical default: `false`	Flag to automatically fetch the output streams from the remote execution

☐	.SSHClient	Struct (See Table 8)	Option to specify the SSH client
☐	.CheckRequirements	Logical default: true	Flag to check if all the requirements for dispatched computations (*e.g.*, key-based authenticated SSH connection, remote utilities) are met. If the check fails, the creation of the DISPATCHER also fails.
☐	.AutoSave	Logical default: true	Flag to save the DISPATCHER object when it is first created to a file in the current working directory. The DISPATCHER object will have no JOB associated with it. The file is read when the command uq_loadDispatcher is called without an argument.

3.2.1 SSH Client

Table 8: DispatcherOpts.SSHClient

☐	.Name	String default: OS-dependent	Name of the SSH client
		'PuTTY'	Default client for Windows operating systems
		'OpenSSH'	Default client for Linux and macOS X operating systems
☐	.SecureConnect	String default: Client-dependent	Utility program to log in to a remote machine and execute commands on the remote machine
		'plink'	Default program for PuTTY
		'ssh'	Default program for OpenSSH
☐	.SecureConnectArgs	String default: Client-dependent	Additional arguments to call either plink or ssh
		'-ssh -T'	Additional arguments for plink
		'-T'	Additional arguments for pscp
☐	.SecureCopy	String default: Client-dependent	Utility program for SSH-based secure copy protocols
		'pscp'	Default program for PuTTY
		'scp'	Default program for OpenSSH
☐	.SecureCopyArgs	String default: Client-dependent	Additional arguments to call either pscp or scp

		''	Additional arguments for `plink`
		''	Additional arguments for `ssh`
☐	`.Location`	String default: ''	Location of the SSH client utility programs (`plink`/`ssh` or `pscp`/`scp`) in the client machine. By default, it is assumed that the programs are callable from the system `PATH`.
☐	`.MaxNumTrials`	Integer scalar default: 5	Number of attempts to send commands or copy files to the remote machine before failing

3.3 Accessing the results

3.3.1 DISPATCHER Objects

Table 9: `myDispatcher = uq_map(...)`

`.Name`	String	Name of the DISPATCHER object
`.Type`	String	Type of the DISPATCHER object
`.Options`	Struct (See Table 7)	Options that were defined in `DispatcherOpts` variable (Section 3.2)
`.Profile`	String	Name of the remote machine profile used to create the DISPATCHER object
`.isExecuting`	Logical	Flag that indicates whether the DISPATCHER object is being used in a remote machine
`.AddToPath`	Cell array	List of paths to add to the remote machine environment
`.AddTreeToPath`	Cell array	List of paths (including their subdirectories) to add to the remote machine environment
`.NumProcs`	Integer scalar	Number of parallel processes used to execute dispatched computations
`.LocalStagingLocation`	String	Location in the local client where temporary files of the dispatch package are created before being dispatched to the remote machine
`.RemoteLocation`	String	Location in the remote machine to which the dispatch package is sent
`.ExecMode`	String	Execution mode of the dispatched computation

Continued on next page

Table 9–continued from previous page

`.SyncTimeOut`	Double scalar	Maximum time used in a synchronous execution
`.JobWallTime`	Double scalar	Maximum duration of executing a remote job (in minutes)
`.Internal`	Struct (See Table 10)	Internal fields
`.Jobs`	JOB array	Array that stores the JOB objects associated with dispatched computations. A JOB object is created and appended to this array every time a dispatcher-aware command is called.

3.3.2 `Internal` Fields

Table 10: `myDispatcher.Internal`

`.Display`	Integer scalar	Level of information displayed during dispatched computation
`.ProfileFullPath`	Cell array	Full path of the profile file in parts
`.RemoteConfig`	Struct (see Table 4)	Specified variables in the profile file
`.SSHClient`	Struct (see Table 8)	Specified option of the SSH client
`.NumCPUs`	Integer scalar	Number of requested CPUs on the remote machine used to execute dispatched computations
`.NumNodes`	Integer scalar	Number nodes used to distribute the computation. It is assumed that all nodes have the same number of CPUs and that all of them together sum up to `.NumCPUs`.
`.CPUsPerNode`	Integer scalar	Number of CPUs available in each node. The number of nodes is computed automatically from this value. If both this option and `.NumNodes` are given, thisoption is ignored.
`.CheckInterval`	Double scalar	Elapsed time (in seconds) before another attempt to update the status of a JOB is made
`.FetchStreams`	Logical	Flag to automatically fetch the output streams from the remote execution
`.RemoteFiles`	Struct (See Table 11)	Default filenames used in the dispatch package

Continued on next page

Table 10–continued from previous page

.RemoteSep	String	Directory separator of the remote machine ('/')

Several files are created as part of the dispatch package and they are sent to the remote machine. The default names for these files are listed in Table 11.

Table 11: myDispatcher.Internal.RemoteFiles

.JobObject	String default: 'JobObj.mat'	Name of the MAT-file that stores the current JOB object
.MATLAB	String default: 'uq_remote_script.m'	Name of the MATLAB script
.Bash	String default: 'uq_remote_script.sh'	Name of the Bash script
.UQLabSession	String default: 'uq_tmp_session.mat'	Name of the saved UQLab session file
.MPI	String default: 'mpifile.sh'	Name of the MPI script
.JobScript	String default: 'qfile.sh'	Name of the job script
.MPIRunPID	String default: 'mpirun.pid'	Name of the file that stores PID of mpirun command
.JobScriptStdOut	String default: 'uq_runtime.out'	Name of the file that stores redirected standard output from a job submission
.SetPATH	String default: 'uq_remote_setpath.sh'	Name of Bash script to set up the remote environment
.Data	String default: 'uq_tmp_data'	Name of the file that stores data for the remote computation
.Output	String default: 'uq_tmp_out'	Name of the file that stores the results (output) of the remote computation
.LogSubmit	String default: '.uq_job_submitted'	Name of the logfile that indicates that the job has been submitted to the remote machine
.LogStart	String default: '.uq_job_started'	Name of the logfile that indicates that the submitted job has been launched
.LogError	String default: '.uqProc_%s_ExecErr'	Name of the logfile that indicates that a process of the remote execution has thrown an error

Continued on next page

Table 11–continued from previous page

`.LogCompleted`	String default: `'.uqProc_%s_ExecCpl'`	Name of the logfile that indicates that a process of the remote execution has been complete

3.3.3 JOB Objects

A JOB object is automatically created every time a dispatcher-aware command is called with a DISPATCHER object. It contains a detailed information to carry out the remote execution. A newly created object is appended to the `Jobs` property (an object array) of the corresponding DISPATCHER object. An individual JOB object can be accessed from the object array using its index (`idx` in the table below).

Table 12: `myDispatcher.Jobs(idx)`

`.Name`	String	Name of the JOB
`.RemoteFolder`	String	Directory on the remote machine where the JOB is located
`.Status`	String	Status of the JOB. See Section 1.3.3 for details.
`.JobID`	String	Scheduler-issued identification of the JOB. If no scheduler is used, `JobID` is the process identifier (PID).
`.ExecMode`	String	Execution mode of the JOB (*i.e.*, dispatched computation)
`.AttachedFiles`	Cell array	List of files and folders on the local client copied and made available on the remote machine
`.AddToPath`	Cell array	List of paths on the remote machine added to the remote machine environment specific to the JOB
`.AddTreeToPath`	Cell array	List of paths on the remote machined (including their subdirectories) added to the remote machine specific to the JOB
`.Tag`	String	Descriptive text for the JOB
`.SubmitDateTime`	String	Date and time when the JOB is submitted to the remote machine (in UTC format)
`.StartDateTime`	String	Date and time when the JOB is started (*i.e.*, launched) in the remote machine (in UTC format)

Continued on next page

Table 12–continued from previous page

`.FinishDateTime`	String	Date and time when the JOB is finished in the remote machine (in UTC format)
`.LastUpdateTime`	String	Date and time when the status of the JOB is updated (in UTC format)
`.QueueDuration`	String	Duration of the JOB in queue if a scheduler is used
`.RunningDuration`	String	Duration of the JOB is running
`.Fetch`	Cell array	List of files on the remote machine that contains the results of the computation. The files will be copied back to the local client machine when the results are fetched.
`.Parse`	Function handle	Function to read the fetched files
`.Merge`	Function handle	Function to merge results of the remote computation in the local client
`.MergeParams`	Struct	Additional parameters for merging results in the local client. These depend on the implementation of the function to merge (*i.e.*, `.Merge`).
`.WallTime`	Double scalar	Wall time specification for the scheduler. This option applies only if a scheduler is used.
`.Data`	Struct (see Table 13)	Data specific to the JOB
`.Task`	Struct (see Table 14)	Details on the remote computation of the JOB
`.FetchStreams`	Logical	Flag to bring the output streams of the remote execution (*i.e.*, standard output and standard error) back to the local MATLAB session. The output streams are stored in the JOB object (*i.e.*, `.OutputStreams`). If the JOB fails, then the output streams will be fetched automatically.
`.OutputStreams`	Struct (see Table 15)	Output streams specific to the JOB

The `Data` field of the JOB object depends on the particular implementation of the dispatcher-aware commands. The contents of `Data` field will be made available in the remote machine.

Table 13 provides the contents of data used in the currently available dispatcher-aware commands.

Table 13: `myDispatcher.Jobs(idx).Data`

`.Inputs`	Any MATLAB data type	Inputs of the remote computation
`.Parameters`	Any MATLAB data type	Parameters used in the remote computation
`.Model`	Any MATLAB data type	Function or UQLAB MODEL objects used in the remote computation
`.SeqGenParameters`	Struct	Parameters used to generate sequence in the remote MATLAB session (only applicable for dispatched `uq_map`)

The `Task` field of the JOB object depends on the particular implementation of the dispatcher-aware commands. Table 14 provides the contents of JOB-specific task used in the currently available dispatcher-aware commands.

Table 14: `myDispatcher.Jobs(idx)Task`

`.NumOfOutArgs`	Integer scalar	Number of output argument to get from the results of the computation in the remote machine (only applicable for dispatched `uq_map`)
`.MatrixMapping`	String	Way to take an element from a matrix (only applicable for dispatched `uq_map`)
`.ExpandCell`	Logical	Flag to expand the content of a cell into a comma-separated list (only applicable for dispatched `uq_map`)
`.ErrorHandler`	Logical or function handle	Error handler for the dispatched computation (only applicable for dispatched `uq_map`)
`.Type`	String	Type of task
`.MATLAB`	Logical	Flag to use MATLAB in the dispatched computation
`.UQLab`	Logical	Flag to load UQLAB in the remote MATLAB session
`.SaveUQLabSession`	Logical	Flag to save the current local UQLAB session and load it in the remote MATLAB session
`.Command`	String or function handle	Command associated with the JOB
`.NumTasks`	Integer scalar	Number of tasks within a JOB.

Continued on next page

Table 14–continued from previous page

`.NumProcs`	Integer scalar	Number of parallel processes in the remote machine. This number differs from the number of CPUs if the number of tasks is smaller than the number of CPUs.

The `OutputStreams` field of the JOB object stores the output streams of the remote execution (*i.e.*, standard output and standard error). Table 15 provides the contents of the field. The contents of each fields are cell arrays that store the output streams line by line.

Table 15: `myDispatcher.Jobs(idx).OutputStreams`

`.SubmitStdOut`	Cell array	Standard output from the JOB submission command
`.JobStdOut`	Cell array	Standard output from the JOB execution
`.JobStdError`	Cell array	Standard error from the JOB execution
`.ProcessStdErr`	Cell array	Standard error from each process of the JOB execution
`.TaskStdOut`	Cell array	Standard output from each task of the JOB (only available in non-MATLAB remote computations)
`.TaskStdErr`	Cell array	Standard error from each task of the JOB (only available in non-MATLAB remote computations)
`.TaskExitStatus`	Cell array	Exit status from each task of the JOB (only available in non-MATLAB remote computations)

3.4 Print a DISPATCHER object

Contextually relevant information for a given DISPATCHER object can be printed using the `uq_print` command.

Syntax

```
uq_print(DISPATCHEROBJ)
uq_print(DISPATCHEROBJ,JOBIDX)
```

`uq_print(DISPATCHEROBJ)` prints a report of the configuration options of the DISPATCHER object DISPATCHEROBJ. The report also includes the information related to the last created JOB object associated with the DISPATCHER object.

uq_print(DISPATCHEROBJ,JOBIDX) prints a report of the configuration options of the DISPATCHER object DISPATCHEROBJ. The report also includes the information related to the associated JOB object selected using its index JOBIDX.

3.5 Save a DISPATCHER object to a file

Syntax

```
uq_saveDispatcher(FILENAME)
uq_saveDispatcher(FILENAME,DISPATCHEROBJ)
uq_saveDispatcher(FILENAME,DISPATCHERIDX)
uq_saveDispatcher(FILENAME,DISPATCHERNAME)
```

uq_saveDispatcher(FILENAME) saves the DISPATCHER object currently selected in the UQLAB session to a MAT-file named FILENAME. If not given, then the file extension is automatically provided. If given, the extension must be .mat.

uq_saveDispatcher(FILENAME,DISPATCHEROBJ) saves a DISPATCHER object DISPATCHEROBJ to a MAT-file named FILENAME.

uq_saveDispatcher(FILENAME,DISPATCHERIDX) saves the DISPATCHER object selected using its index DISPATCHERIDX from the current UQLAB session to a MAT-file named FILENAME.

uq_saveDispatcher(FILENAME,DISPATCHERNAME) saves the DISPATCHER object selected using its name DISPATCHERNAME from the current UQLAB session to a MAT-file named FILENAME.

3.6 Load a DISPATCHER object from a file

Syntax

```
uq_loadDispatcher
uq_loadDispatcher(FILENAME)
uq_loadDispatcher(..., 'NoDuplicate', false)
DISPATCHEROBJ = uq_loadDispatcher(FILENAME)
DISPATCHEROBJ = uq_loadDispatcher(FILENAME,'-private')
```

uq_loadDispatcher loads a DISPATCHER object stored in an autosaved MAT-file located in the current working directory and adds the object to the current UQLAB session. The MAT-file is, by default, automatically created with a special filename when a DISPATCHER object is created. When there are multiple of such files, the most recent one is automatically loaded.

`uq_loadDispatcher(FILENAME)` loads a DISPATCHER object stored in a MAT-file named FILENAME and adds the object to the current UQLAB session. If the `.mat` extension is not given in FILENAME, it will be added. If a DISPATCHER object of the same name already in the current UQLAB session, the new object is loaded with a modified name.

`uq_loadDispatcher(FILENAME,DISPATCHEROBJ)` loads a DISPATCHER object stored in a MAT-file and only adds the object to the current UQLAB session if there is no DISPATCHER object of the same name already in the session.

`uq_loadDispatcher(FILENAME,DISPATCHERIDX)` additionally returns the DISPATCHER object in a variable DISPATCHEROBJ.

`uq_loadDispatcher(FILENAME,DISPATCHERNAME)` loads a DISPATCHER object stored in FILENAME and returns the object DISPATCHEROBJ. The DISPATCHER object will not be imported to the current UQLAB session.

3.7 Interact with a JOB object

UQLAB offers various commands to help users interact with JOB objects associated with a DISPATCHER object. The available helper commands range from a command to get a JOB status to delete a JOB.

3.7.1 Listing JOBS

Syntax

```
uq_listJobs(DISPATCHEROBJ)
uq_listJobs(DISPATCHEROBJ,JOBIDX)
uq_listJobs(..., 'UpdateStatus', true)
```

`uq_listJobs(DISPATCHEROBJ)` lists all the JOBS associated with the DISPATCHER object DISPATCHEROBJ. By default, the status of each Job on the remote machine will not be updated.

`uq_listJobs(DISPATCHEROBJ,JOBIDX)` lists the Jobs associated with DISPATCHEROBJ selected by their index JOBIDX. By default, JOBIDX is `'-all'`; that is, all the associated JOBS are selected.

`uq_listJobs(DISPATCHEROBJ, NAME, VALUE)` lists the JOBS selected by `uq_findJobs` helper function (see Section 3.7.6).

`uq_listJobs(..., 'UpdateStatus', true)` list the JOBS associated with DISPATCHEROBJ after updating each of their status on the remote machine.

3.7.2 Getting the status of a JOB

Syntax

```
JOBSTATUS = uq_getStatus(DISPATCHEROBJ)
JOBSTATUS = uq_getStatus(DISPATCHEROBJ,JOBIDX)
JOBSTATUS = uq_getStatus(DISPATCHEROBJ,JOBIDC)
JOBSTATUS = uq_getStatus(DISPATCHEROBJ,'-all')
JOBSTATUS = uq_getStatus(DISPATCHEROBJ, NAME, VALUE)
JOBSTATUS = uq_getStatus(DISPATCHEROBJ, 'Update', false)
[JOBSTATUS,JOBSTATUSID] = uq_getStatus(...)
```

`JOBSTATUS = uq_getStatus(DISPATCHEROBJ)` gets the status of the last created Job in the `DISPATCHEROBJ`. The status will automatically be updated with the current state of the JOB in the remote machine. To see the different possible status, see Section 1.3.3.

`JOBSTATUS = uq_getStatus(DISPATCHEROBJ,JOBIDX)` gets the status of the JOB selected using its index `JOBIDX`.

`JOBSTATUS = uq_getStatus(DISPATCHEROBJ,JOBIDC)` gets the status of multiple JOBs selected using their indices `JOBIDC` given as a vector.

`JOBSTATUS = uq_getStatus(DISPATCHEROBJ,'-all')` gets the status of all JOBs associated with the DISPATCHER object `DISPATCHEROBJ`.

`JOBSTATUS = uq_getStatus(DISPATCHEROBJ, NAME, VALUE)` gets the status of of JOBs selected by `uq_findJobs` helper function (see Section 3.7.6).

`JOBSTATUS = uq_getStatus(DISPATCHEROBJ, 'UpdateStatus', false)` gets the status of JOBs without updating it based on the current state of the JOBs in the remote machine.

`[JOBSTATUS,JOBSTATUSID] = uq_getStatus(...)` also returns the integer identification number (ID) of the status. The identification numbers are:

- -1: `'failed'`
- 0: `'canceled'`
- 1: `'pending'`
- 2: `'submitted'`
- 3: `'running'`
- 4: `'complete'`

Note: Updating the Status of a JOB(s) will modify the DISPATCHER object in-place (*i.e.*, as a side effect).

3.7.3 Updating the status of a JOB

Syntax

```
uq_updateStatus(DISPATCHEROBJ)
uq_updateStatus(DISPATCHEROBJ,JOBIDX)
uq_updateStatus(DISPATCHEROBJ,JOBIDC)
uq_updateStatus(DISPATCHEROBJ,'-all')
```

`uq_updateStatus(DISPATCHEROBJ)` updates the status of the last created JOB associated with a DISPATCHER object `DISPATCHEROBJ` based on the current of the JOB in the remote machine. The update on `DISPATCHEROBJ` happens in-place (*i.e.*, as a side effect).

`uq_updateStatus(DISPATCHEROBJ,JOBIDX)` updates the status of the JOB selected using its index `JOBIDX`.

`uq_updateStatus(DISPATCHEROBJ,JOBIDC)` updates the status of multiple JOBs selected by their indices `JOBIDC` given as a vector.

`uq_updateStatus(DISPATCHEROBJ,'-all')` updates the status of all JOBs associated with a DISPATCHER object `DISPATCHEROBJ`.

Note: Updating the Status of a JOB(s) will modify the DISPATCHER object in-place (*i.e.*, as a side effect).

3.7.4 Waiting for a JOB

Syntax

```
uq_waitForJob(DISPATCHEROBJ)
uq_waitForJob(DISPATCHEROBJ,JOBIDX)
uq_waitForJob(..., NAME, VALUE)
WAITSTATUS = uq_waitForJob(...)
```

`uq_waitForJob(DISPATCHEROBJ)` waits for the last created JOB associated with the DISPATCHER object `DISPATCHEROBJ` finish.

`uq_waitForJob(DISPATCHEROBJ,JOBIDX)` waits for a JOB associated with `DISPATCHEROBJ` selected using its index `JOBIDX` (a scalar) to finish. Only one JOB can be waited for at time.

`uq_waitForJob(..., NAME, VALUE)` waits for a JOB with additional (optional) `NAME/VALUE` argument pairs (see Table 16).

`WAITSTATUS = uq_waitForJob(...)` returns if the specified JOB status has actually actually been reached at the end the wait.

Table 16: `uq_waitForJob(..., NAME, VALUE)`

`'JobStatus'`	Integer scalar default: 4	Status of the JOB to reach
`'WaitTimeOut'`	Double scalar default: `.Internal.Timeout`	Maximum time to wait (in seconds). The default value is stored in the DISPATCHER object.
`'CheckInterval'`	Double scalar default: `.Internal.CheckInterval`	Time interval to check the JOB status in the remote machine (in seconds). The default value is stored in the DISPATCHER object.

3.7.5 Fetching the results of a JOB

Syntax

```
RESULTS = uq_fetchResults(DISPATCHEROBJ)
RESULTS = uq_fetchResults(DISPATCHEROBJ,JOBIDX)
RESULTS = uq_fetchResults(..., NAME, VALUE)
```

`RESULTS = uq_fetchResults(DISPATCHEROBJ)` fetches the results of the remote computation of the last created JOB associated with the DISPATCHER object `DISPATCHEROBJ`. `RESULTS` depends on the JOB.

`RESULTS = uq_fetchResults(DISPATCHEROBJ,JOBIDX)` fetches the results of the remote computation of a JOB associated with the `DISPATCHEROBJ` selected using its index `JOBIDX`.

`RESULTS = uq_fetchResults(..., NAME, VALUE)` fetches the results of the remote computation with additional (optional) `NAME/VALUE` arguments (see Table 17).

Table 17: `uq_fetchResults(..., NAME, VALUE)`

`'KeepFiles'`	Logical default: false	Flag to keep the fetched files
`'DestDir'`	String default: `.LocalStagingLocation`	Destination directory in the local client for the fetched files. The default is based on the value stored in the DISPATCHER object and a new directory with the JOB name will be created inside.
`'ForceFetch'`	Logical default: `false`	Flag to force the fetching; it will attempt to fetch the results from the available files. If the results are force fetched, then merge operation is skipped.

Continued on next page

Table 17–continued from previous page

'SrcDir'	String default: ''	Source directory in the local client. The argument is used when the results have been previously fetched into a 'DestDir' and the files have been kept (*i.e.*, 'KeepFiles' is true).

3.7.6 Finding JOBs

Syntax

```
JOBIDX = uq_findJobs(DISPATCHEROBJ)
JOBIDX = uq_findJobs(DISPATCHEROBJ, NAME, VALUE)
JOBIDX = uq_findJobs(..., 'Operator', 'Or')
[JOBIDX,JOB] = uq_findJobs(...)
```

`JOBIDX = uq_findJobs(DISPATCHEROBJ)` gets all indices `JOBIDX` of the JOBs associated with a DISPATCHER object `DISPATCHEROBJ`.

`JOBIDX = uq_findJobs(DISPATCHEROBJ, NAME, VALUE)` finds the JOBs associated with `DISPATCHEROBJ` whose properties matches the properties specified as `NAME/VALUE` argument pairs and returns the indices `JOBIDX` (see Table 18).

`JOBIDX = uq_findJobs(..., 'Operator', 'Or')` finds JOBs that matches the property criteria specified by `NAME/VALUE` argument pairs combined with `'Or'` operator. By default, the criteria are matched by `'And'` operator.

`[JOBIDX,JOB] = uq_findJobs(...)` additionally returns the JOB objects that match the specified property criteria.

Table 18: `uq_findJobs(..., NAME, VALUE)`

'Name'	String default: ''	Name of the JOBs (if specified, regular expression matching is supported)
'Status'	String or integer scalar default: ''	Status of the JOBs (if specified, it must be exact to match)
'JobID'	String default: ''	ID of the JOBs (if specified, it must be exact to match)
'Tag'	String default: ''	Tag (descriptive text) of the JOBs (if specified, regular expression matching is supported)
'ExecMode'	String default: ''	Execution mode of the JOBs (if specified, use either 'sync' or 'async')

Continued on next page

Table 18–continued from previous page

'SubmitDateTime'	String default: ''	Date and time of the JOBs submitted (if specified, regular expression matching is supported)
'StartDateTime'	String default: ''	Date and time of the JOBs started (if specified, regular expression matching is supported)
'String'	String default: ''	Date and time of the JOBs finished (if specified, regular expression matching is supported)
'RunningDuration'	String default: ''	Running duration of the JOBs (if specified, regular expression matching is supported)
'QueueDuration'	String default: ''	Queue duration of the JOBs (if specified, regular expression matching is supported)

3.7.7 Canceling a JOB

Syntax

```
uq_cancelJob(DISPATCHEROBJ,JOBIDX)
```

`uq_cancelJob(DISPATCHEROBJ,JOBIDX)` cancels a JOB in a DISPATCHER object `DISPATCHEROBJ`. The JOB is selected using its index `JOBIDX` (a scalar) and it must either be submitted or running. Only one JOB can be canceled at a time.

Note: `uq_cancelJob` executes a scheduler-specific job cancelation command on the ID of the selected JOB to the remote machine. If scheduler is used, then the cancelation command must be set up properly. Otherwise, `uq_cancelJob` sends a kill signal for the corresponding `mpirun` process instead.

3.7.8 Deleting a JOB

Syntax

```
uq_deleteJob(DISPATCHEROBJ,JOBIDX)
uq_deleteJob(DISPATCHEROBJ,JOBIDC)
```

`uq_deleteJob(DISPATCHEROBJ,JOBIDX)` deletes a JOB in a DISPATCHER object `DISPATCHEROBJ` selected using its index `JOBIDX` (a scalar). Deleting a JOB removes all the remote files associated with the JOB as well as the JOB itself from `DISPATCHEROBJ`. Only a finished JOB can be deleted.

`uq_deleteJob(DISPATCHEROBJ,JOBIDC)` deletes multiple JOBS in a DISPATCHER object `DISPATCHEROBJ` selected by their indices `JOBIDC` (a vector).

Note: Only finished JOBS (*i.e.*, JOBS with `'complete'`, `'canceled'`, or `'failed'` status) can be deleted.

3.7.9 Recreating a JOB

Syntax

```
uq_recreateJob(DISPATCHEROBJ)
uq_recreateJob(DISPATCHEROBJ,JOBIDX)
uq_recreateJob(..., NAME, VALUE)
```

`uq_recreateJob(DISPATCHEROBJ)` recreates the last created JOB in a DISPATCHER object `DISPATCHEROBJ` as a new JOB appended to last entry of the `Jobs` array.

`uq_recreateJob(DISPATCHEROBJ,JOBIDX)` recreates a JOB in `DISPATCHEROBJ` selected using its index `JOBIDX`. Only one JOB can be recreated at a time.

`uq_recreateJob(..., NAME, VALUE)` recreates a JOB in `DISPATCHEROBJ` and overrides some of the previous properties of the JOB specified using `NAME/VALUE` argument pairs (see Table 19).

Note: Recreated JOB is not automatically submitted (see Section 3.7.10 to submit a JOB).

In the table below `JobObj` refers to `DispatcherObj.Jobs(jobIdx)`. Note that some properties can be override after the new JOB has been created. This includes: `Tag`, `ExecMode`, `Parse`, and `Merge`.

Table 19: `uq_recreateJob(..., NAME, VALUE)`

`'Name'`	String default: `JobObj.Name`	Name of the JOB
`'RemoteFolder'`	String default: `JobObj.RemoteFolder`	Directory in the remote machine to store JOB-specific files
`'ExecMode'`	String default: `JobObj.ExecMode`	Execution mode of the JOB
`'AttachedFiles'`	Cell array default: `JobObj.AttachedFiles`	List of attached files and folders to send to the remote machine
`'AddToPath'`	Cell array default: `JobObj.AddToPath`	List of directories in the remote machine to add to the PATH of the remote environment

Continued on next page

Table 19–continued from previous page

`'AddTreeToPath'`	Cell array default: `JobObj.AddTreeToPath`	List of directories (including, their subdirectories) to add to the PATH of the remote environment
`'Tag'`	String default: `JobObj.Tag`	Descriptive text for the JOB
`'Fetch'`	Cell array default: `JobObj.Fetch`	List of files to fetch from the remote machine
`'Parse'`	Function handle default: `JobObj.Parse`	Function handle to parse fetched files
`'Merge'`	Function handle default: `JobObj.Merge`	Function handle to merge fetched results
`'Data'`	Struct default: `JobObj.Data`	Data (inputs, parameters, etc.) associated with the JOB (see Table 13)
`'Task'`	Struct default: `JobObj.Task`	Task specific to the JOB (see Table 14)
`'WallTime'`	Double scalar default: `JobObj.WallTime`	Wall time for the remote scheduler (in minutes)
`'FetchStreams'`	Logical default: `JobObj.FetchStreams`	Flag to automatically fetch the output streams of the remote execution

3.7.10 Submitting a JOB

Syntax

```
uq_submitJob(DISPATCHEROBJ,JOBIDX)
```

`uq_submitJob(DISPATCHEROBJ,JOBIDX)` submits a JOB in a DISPATCHER object `DISPATCHEROBJ` selected using its index `JOBIDX` to the remote machine. Only one JOB can be submitted at a time and its status must be `'pending'`. Submitting a Job modifies the selected JOB in `DISPATCHEROBJ` in-place.

Note: Only `'pending'` JOB can be submitted. Submitting a JOB modifies the selected JOB in the DISPATCHER object in-place (*i.e.*, as a side effect).

3.7.11 Retrieving JOBs from the remote machine

Syntax

```
uq_retrieveJobs(DISPATCHEROBJ)
```

`uq_retrieveJobs(DISPATCHEROBJ)` retrieves JOBs from a remote location specified in a DISPATCHER object `DISPATCHEROBJ` and adds it to the DISPATCHER object. If an identical JOB is already in `DISPATCHEROBJ`, it will not be added. The JOB status is automatically updated before it is added to `DISPATCHEROBJ`.

`uq_retrieveJobs(DISPATCHEROBJ,DIRNAMES)` retrieves JOBs from a remote location specified in a DISPATCHER object `DISPATCHEROBJ` with the JOB-specific directory names given in a cell array `DIRNAMES`.

Note:
- The retrieving process looks for the sub-directories (one level) within the remote location. By default, the JOB-specific remote directory name is based on the timestamp of its creation. The JOBs will be retrieved from the oldest to the newest.
- Retrieving JOBs modifies the DISPATCHER object in-place (*i.e.*, as a side effect).
- Depending on the verbosity level of the Dispatcher object, additional information may be displayed on the MATLAB Command Window during retrieval process.

Appendix A

Setting up a secure SSH key-based authentication

The first step in setting up a DISPATCHER object is to make sure a *secure* SSH connection to the remote machine can be established without the need of prompting for a password, through key-based authentication. This allows secure and seamless communication from the local client to the remote machine (*e.g.*, sending dispatch package, getting the status of a remote computation); users do not need to authenticate themselves repeatedly using a password. Note that all communications are still securely encrypted at industry standards, but the authentication credentials are stored in a securely encrypted file.

Key-based authentication is created using the RSA[1] public and private key scheme[2].

The process of establishing an SSH connection to the remote machine using a key-based authentication is operating-system-dependent. The steps are explained below for Windows (Section A.1) and Linux/macOS (Section A.2) operating systems.

A.1 Windows

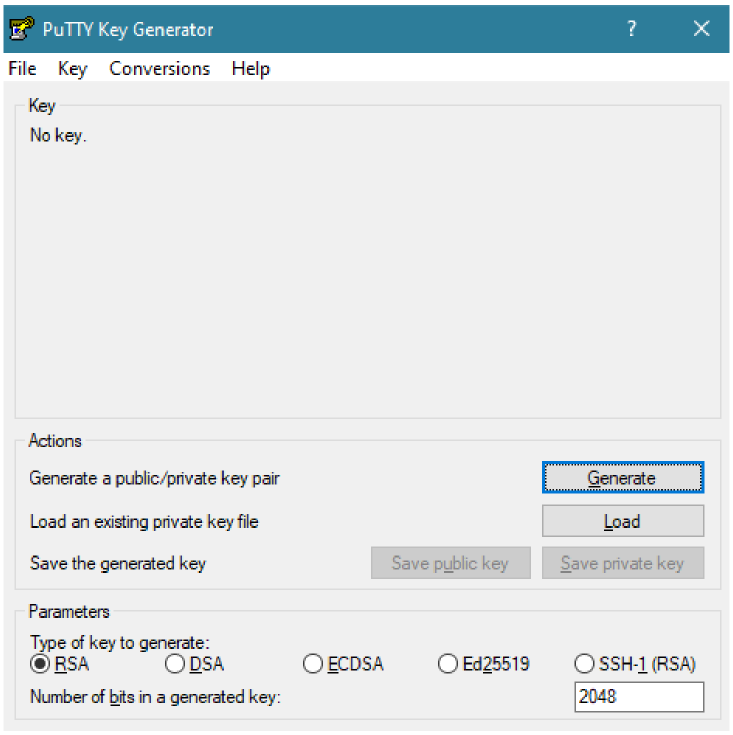
The open source PuTTY SSH client is required for a local client running a Windows operating system. Please follow the installation instructions on the PuTTY website (https://putty.org/) before proceeding.

Once the PutTTY client is installed, an SSH key-based authentication on a target remote machine can be set up as follows:

Step 1: Generate SSH key pairs

1. Open the `PuTTYgen` application (part of the PuTTY installation). The main screen of the application is shown in Figure 11.

[1]Rivest–Shamir–Adleman
[2]the public and private keys are collectively referred to as the SSH key-pair


PuTTY Key Generator

File Key Conversions Help

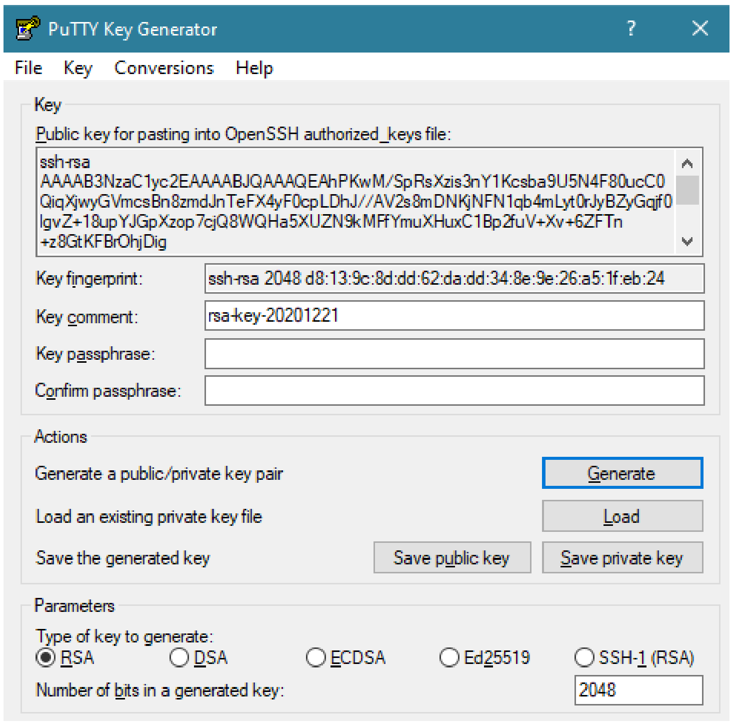
Key
No key.

Actions
Generate a public/private key pair — Generate
Load an existing private key file — Load
Save the generated key — Save public key — Save private key

Parameters
Type of key to generate:
[✓ RSA] DSA ECDSA Ed25519 SSH-1 (RSA)
Number of bits in a generated key: 2048

Figure 11: The main screen of `PuTTYgen`.

2. Click the `Generate` button. You will be asked to move around the mouse cursor around the blank zone. Once finished, the main screen of the application will look similar to Figure 12.


PuTTY Key Generator

File Key Conversions Help

Key
Public key for pasting into OpenSSH authorized_keys file:

```
ssh-rsa
AAAAB3NzaC1yc2EAAAABJQAAAQEAhPKwM/SpRsXzis3nY1Kcsba9U5N4F80ucC0
QiqXjwyGVmcsBn8zmdJnTeFX4yF0cpLDhJ//AV2s8mDNKjNFN1qb4mLyt0rJyBZyGqjf0
lgvZ+18upYJGpXzop7cjQ8WQHa5XUZN9kMFfYmuXHuxC1Bp2fuV+Xv+6ZFTn
+z8GtKFBrOhjDig
```

Key fingerprint: ssh-rsa 2048 d8:13:9c:8d:dd:62:da:dd:34:8e:9e:26:a5:1f:eb:24
Key comment: rsa-key-20201221
Key passphrase:
Confirm passphrase:

Actions
Generate a public/private key pair — Generate
Load an existing private key file — Load
Save the generated key — Save public key — Save private key

Parameters
Type of key to generate:
[✓ RSA] DSA ECDSA Ed25519 SSH-1 (RSA)
Number of bits in a generated key: 2048

Figure 12: The main screen of `PuTTYgen` after SSH keys have been generated.

3. Use the "Save private key" function to save the generated key to a desired, preferably secured location on the local client computer.

   Note: Make sure the private key is saved without a passphrase, otherwise the passphrase will be required at every connection.

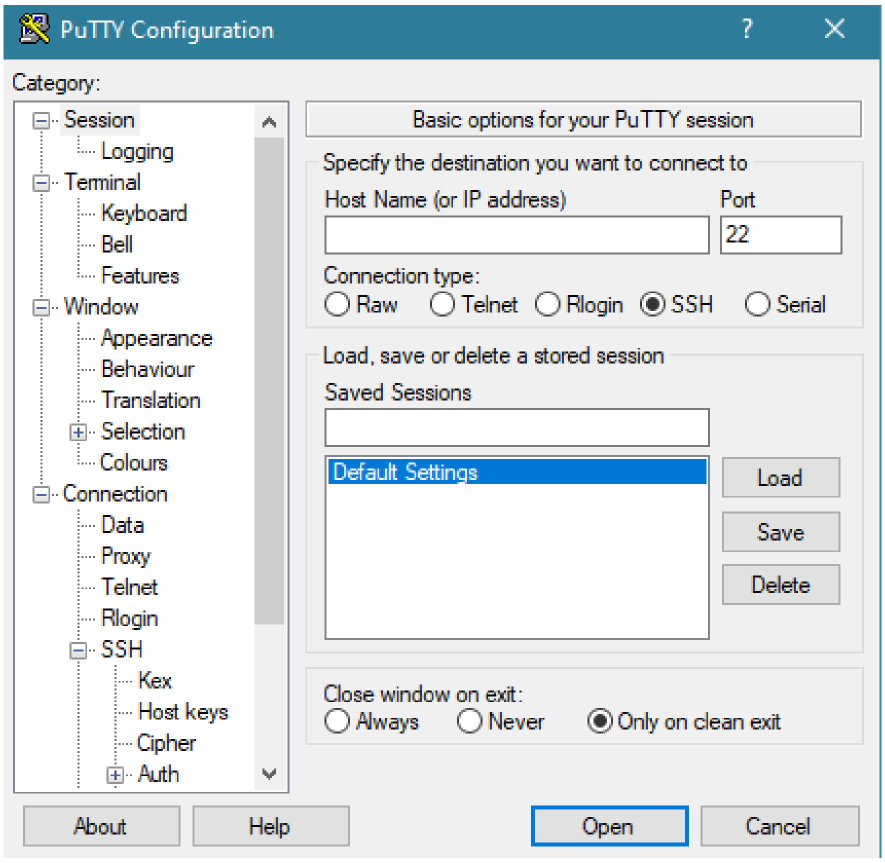
   Never share your private key! Once the public key is registered on the remote machine, anyone with a private key can connect with your credentials.

4. Copy the public key to the clipboard. This is required in the subsequent steps to add the public key to the remote machine.

Note: Please take note of the name and location of the private key file because it will be needed to set up the profile files (Appendix B).

Step 2: Add the public key to the remote machine

1. Open PuTTY (also part of the PuTTY installation). The main screen of the application will look like Figure 13.


PuTTY Configuration

Category:
- Session
  - Logging
- Terminal
  - Keyboard
  - Bell
  - Features
- Window
  - Appearance
  - Behaviour
  - Translation
  - Selection
  - Colours
- Connection
  - Data
  - Proxy
  - Telnet
  - Rlogin
  - SSH
    - Kex
    - Host keys
    - Cipher
    - Auth

Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address) Port 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Load, save or delete a stored session

Saved Sessions

Default Settings

Load Save Delete

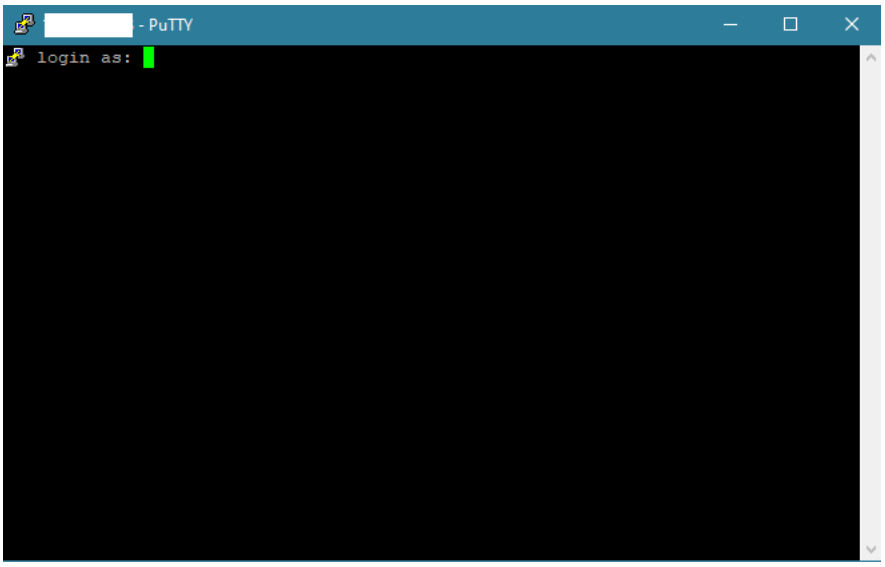
Close window on exit: Always Never [✓ Only on clean exit]

About Help Open Cancel

Figure 13: The main screen of PuTTY.

2. Type the hostname (or IP address) of the remote machine in the "Host Name" box (This

is typically the login node of a remote cluster, or the hostname of a remote workstation). Click `Open` and, if the details are correct, a command window like shown in Figure 14 will open and prompt for username and password. Log in using your credentials.



```
login as:
```

Figure 14: Login into the remote machine using PuTTY.

3. Once logged into the remote machine, open the following file in your preferred text editor:
   `/yourhomefolder/.ssh/authorized_keys`
   (change `yourhomefolder` as appropriate, typically `/home/username`).
   Any editor can be used, below the editor `nano` is used.

   ```
   nano ~/.ssh/authorized_keys
   ```

   The nano editor will open and look similar to Figure 15.

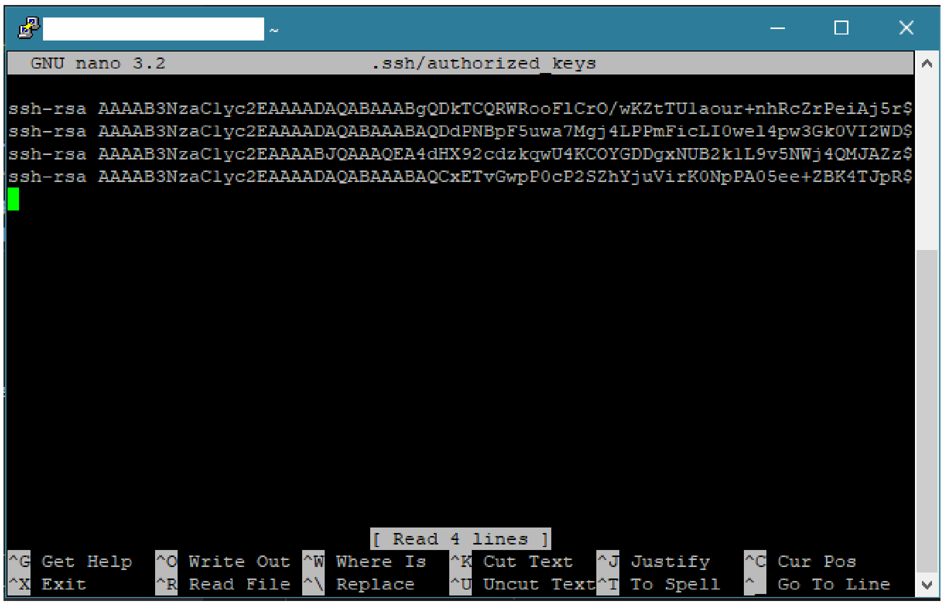
   Note: If a key-pair exchange authentication has never been set up before on the remote machine, the `authorized_keys` file will likely either be empty or not exist at all. Please create it in the latter case.

4. Use the arrow keys to move to a new line at the end of the file, and append your public key by pasting it.

   Note: A text string copied in the clipboard can be pasted on the command window by right clicking on it.

5. Once the public key string is pasted, save the file and exit the editor (*e.g.*, in `nano`, press `Ctrl-O` and enter to save, then `Ctrl-X` to exit the editor).
6. Type `exit` in the command prompt to close the session.

The remote machine is now ready to accept connections made with the private key generated earlier.



```
[redacted] ~
  GNU nano 3.2                        .ssh/authorized_keys

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQDkTCQRWRooFlCrO/wKZtTUlaour+nhRcZrPeiAj5r$
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDdPNBpF5uwa7Mgj4LPPmFicLI0wel4pw3Gk0VI2WD$
ssh-rsa AAAAB3NzaC1yc2EAAAABJQAAAQEA4dHX92cdzkqwU4KCOYGDDgxNUB2klL9v5NWj4QMJAZz$
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCxETvGwpP0cP2SZhYjuVirK0NpPA05ee+ZBK4TJpR$

                              [ Read 4 lines ]
^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell   ^_ Go To Line
```

Figure 15: The file /.ssh/authorized_keys opened in nano.

Optionally, users may create and save a PuTTY session that can connect to the remote machine without prompting for a password. Both the private key directly or a PuTTY saved session can be used by the DISPATCHER to connect to the remote client.

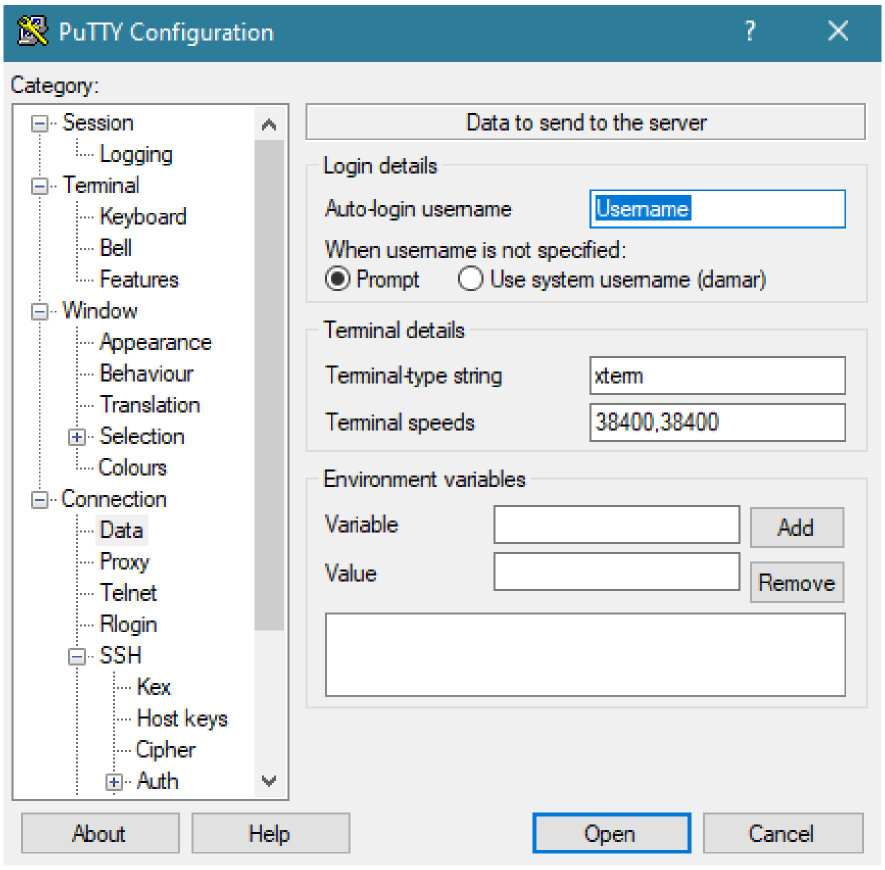
(Optional) Step 3: Save a PuTTY session

1. Open `PuTTY` again
2. Fill in the `Hostname` field as before, with the hostname of the target machine for which the keys have just been created.
3. From the left panel, navigate to `Connection` → `Data`.
4. Fill in the `Auto-login username` field with the username used to log in to the remote machine (Figure 16).

   Note: Do not specify a password! Only the login name is needed

5. From the left panel, navigate to `Connection` → `SSH`.
6. In the `Authentication parameters` panel, click `Browse` and select the private key (`.ppk`) previously saved (Figure 17).
7. From the left panel, navigate now to `Session`. Type a name (e.g. `UQLab`) on the `Saved Sessions` field and click `Save` (Figure 18).

A PuTTY session has been saved, and its name can be used as part of setting up the profile file explained in Appendix B.



Data to send to the server

Login details

Auto-login username: Username

When username is not specified:
[✓ Prompt] Use system username (damar)

Terminal details

Terminal-type string: xterm

Terminal speeds: 38400,38400

Environment variables

Variable:

Value:

Figure 16: Fill in the `Auto-login username` field with the username.

A.2 Linux and macOs

Most Linux distributions and macOS X (a UNIX-based OS) already ship with the OpenSSH client. If this software is missing, install it first using the standard procedure of your operating system before proceeding.

An SSH key-based authentication for a secure connection to a remote machine can be set up directly from the command line as follows:

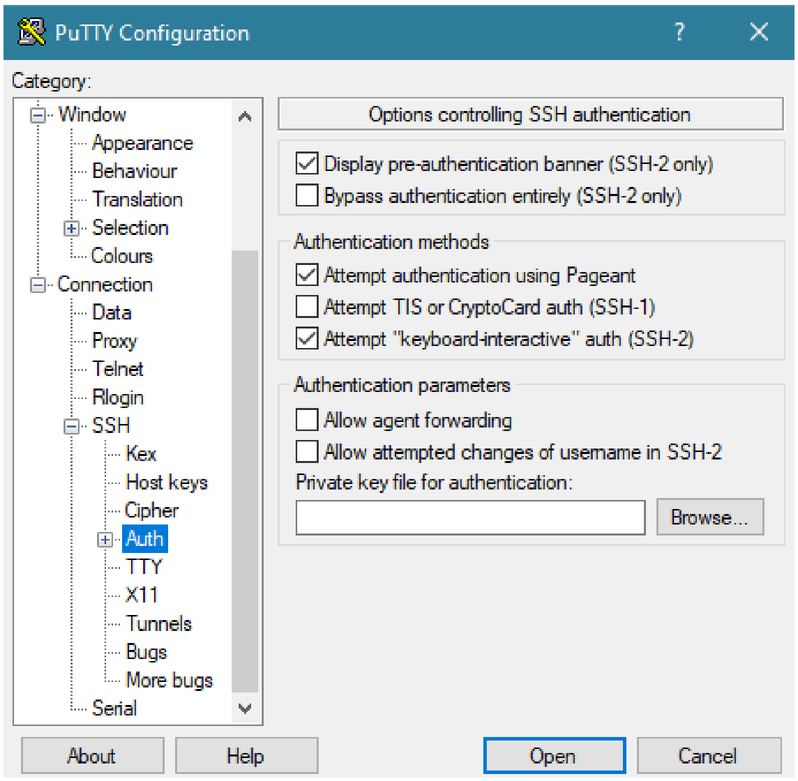
Step 1: Generate SSH key pairs

1. Open a terminal or console application.
2. Generate the keys by typing in the terminal the following command:

```
ssh-keygen
```

Note: If you receive a message similar to "`ssh-keygen: command not found`", you may have to install the OpenSSH client.

3. If the command completes successfully, you will be prompted to `Enter a file in which to save the key`. Enter a desired filename and location. For illustration,



Options controlling SSH authentication

- [✓] Display pre-authentication banner (SSH-2 only)
- [ ] Bypass authentication entirely (SSH-2 only)

Authentication methods

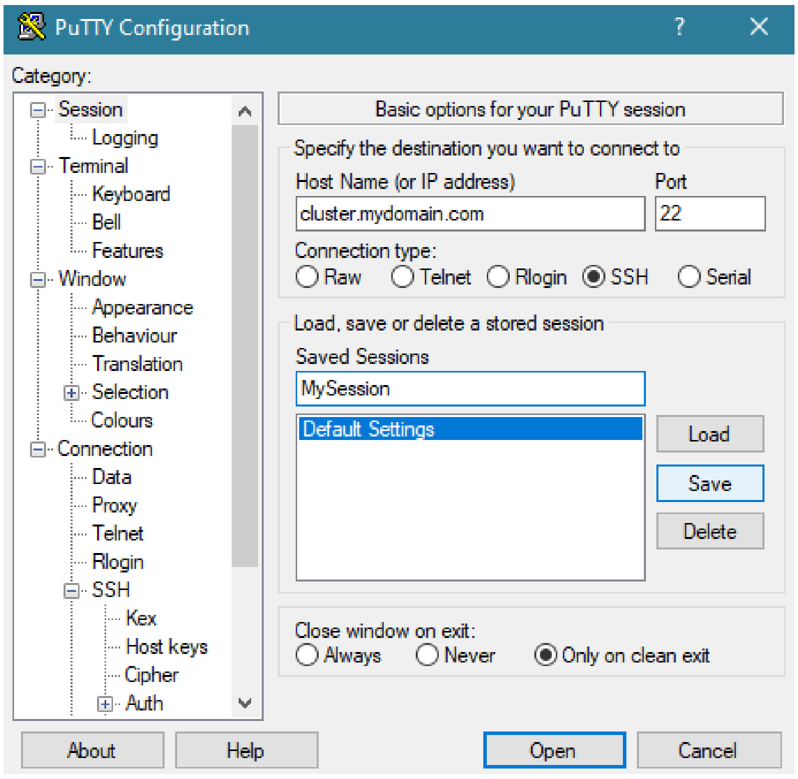
- [✓] Attempt authentication using Pageant
- [ ] Attempt TIS or CryptoCard auth (SSH-1)
- [✓] Attempt "keyboard-interactive" auth (SSH-2)

Authentication parameters

- [ ] Allow agent forwarding
- [ ] Allow attempted changes of username in SSH-2

Private key file for authentication:

Figure 17: Select the previously saved private key to fill in the private key field.



Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address)	Port
cluster.mydomain.com	22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Load, save or delete a stored session

Saved Sessions

MySession

Default Settings

Close window on exit: Always Never [✓ Only on clean exit]

Figure 18: Save the session with a desired name.

suppose the name of they key is `myKey` saved inside the current working directory:

```
Generating public/private rsa key pair.
Enter file in which to save the key (/home/jdoubt/.ssh/id_rsa): myKey
```

4. Skip the passphrase by entering an empty text string.
   The terminal will then display something similar to:

```
Your identification has been saved in MyKey.
Your public key has been saved in MyKey.pub.
The key fingerprint is:
9d:88:89:c0:69:12:95:d1:af:fc:ad:36:60:d5:12:b2
The key randomart image is:
+--[ RSA 2048]----+
|o.+=             |
| =o..            |
|      . + o S o  |
|.. o             |
|.=. .            |
|   .        .o   |
|o =.  .          |
| + . o o E .     |
|   .+o.          |
+-----------------+
```

Step 2: Add the public key to the remote machine

1. Open a terminal or console application.
2. Add the content of the previously created SSH public key (the one with `.pub` extension) to the list of authorized keys of the remote machine by typing (replace `Username` and `Hostname` with the appropriate values):

```
cat myKey.pub | ssh Username@Hostname 'cat >> .ssh/authorized_keys'
```

   The command appends the content of the public key file to the list of authorized keys.
3. If this is the first time a connection is made to the remote machine, you may receive a prompt similar to:

```
The authenticity of host 'hostname (111.111.111)' can't be established.

RSA key fingerprint is 97:8c:1b:f2:6f:14:6b:5c:3b:ec:aa:46:46:74:7c:40.
Are you sure you want to continue connecting (yes/no)?
```

   Enter `yes` to continue.
4. Enter the password to log in to the remote machine when prompted.

Once these steps are completed, an SSH connection to the remote machine using the key-based authentication can be established using the private key. Keep a note on the name and the location of the private key as they are required to set up the profile file, as explained in the next appendix.

Appendix B

Setting up a remote machine profile file

Setting up a remote machine profile file (hereafter, *profile file*) is an essential step for creating a DISPATCHER object (see Section 2.2.1). A profile file is a MATLAB script that contains all the information required to connect to a remote machine. This is required so that the UQLAB DISPATCHER can communicate with the remote machine to dispatch computations and retrieve the corresponding results. Specifically, a profile file contains information about:

- setting up a connection to the remote machine (including *hostname*, *username*, *private key*, etc.);
- remote execution path;
- the required software and related configuration on the remote environment (*e.g.*, MATLAB, UQLAB);
- dispatch infrastructure on the remote machine (e.g. scheduler, environment setup).

The file needs to be created only once for a given remote machine profile.

To assist users in setting up their profile file, UQLAB provides two templates that can be adapted to fit their needs: a basic profile that contains only the necessary information and a more advanced profile that includes less common fine-tuning options. The templates can be found in the `Profiles/HPC` subfolder of the main UQLAB installation folder.

In the subsequent sections, the two templates are further explained in detail. For reference, Table 4-6 in Chapter 3 provide the detailed description, data type, and default value of every variable available in the profile file.

Note: All option names and values in the template files are case sensitive, so please pay close attention to the capitalization of each option.

B.1 Basic template

The basic template contains all the variables required by the DISPATCHER module to dispatch UQLAB computations to many remote machine configurations. It is organized into three different sections: authentication, remote workspace, and remote computing environment.

While the ordering of the variables does not matter, it is good practice to organize them in the profile file.

Note: If multiple variables of the same name are specified in the same profile file, only the last one will be taken into account.

B.1.1 Authentication

The authentication section configures the encrypted login strategy for the specific remote machine.

Below are the template entries for the authentication section:

```
Hostname = 'my.host.name';
Username = 'myusername';
PrivateKey = '/path/to/myPrivateKey';
```

Users may get the values for `Hostname` and `Username` from their system administrator or the distributed computing resources documentation. The value for `PrivateKey` is the SSH private key file used to establish a passwordless SSH connection to the remote machine. See Appendix A for a detailed explanation of how to set up a private key file.

Note: The private key filename in `PrivateKey` must be specified in full, *i.e.*, including its full path.

Instead of providing those three pieces of information, Windows users can alternatively directly specify a PuTTY saved session (see Step 3 in Appendix A.1). In this case, the saved session name can be directly provided as follows:

```
SavedSession = 'mySavedSession';
```

Note that a key-pair based authentication needs to be set up in PuTTY for the DISPATCHER to properly communicate with the remote host. Appendix A.1 (Step 3) provides the detail of the steps required to set it up.

Note: The DISPATCHER module will verify if key-pair SSH authentication to the remote machine can be successfully established and throw an error in case it cannot.

Note: The first set of variables (`Hostname`, `Username`, `PrivateKey`) and `SavedSession` are mutually exclusive. If the user provides both, an error will be thrown when a DISPATCHER object is created due to ambiguous information.

B.1.2 Remote workspace

This section contains a single option: the location of a remote folder where the user has read-write access. For each dispatched computation, the DISPATCHER module uses this location to store the job information and the computation results. The remote folder (directory) can be specified, for example:

```
RemoteFolder = '/home/myusername/myDispatchedComputations';
```

If the remote folder does not exist, the DISPATCHER module will automatically create the folder.

There are several restrictions on how a remote execution folder can be specified:

- The user must have a write-access to the specified remote folder.
- The tilde (~) symbol as a shortcut to `$HOME` is not supported: the user must use the full path to their `$HOME` folder.
- **[Only when using the TORQUE remote scheduler]** The full path of the remote folder must not contain whitespaces.

Note: The DISPATCHER module will verify these restrictions, and throw an error if needed when a DISPATCHER object is created.

B.1.3 Remote computing environment

The remote computing environment section stores the variables used to set up the computing environment on the remote machine. This includes, for example, the location of MATLAB and UQLAB as well as the remote job scheduler. For instance, to specify that the MATLAB executable is available in the `/usr/local/bin` folder, and that the UQLAB installation folder is `/home/myusername/uqlab`, one may write:

```
MATLABCommand = '/usr/local/bin/matlab';
RemoteUQLabPath = '/home/myusername/uqlab';
```

Note: If UQLAB is used during the remote calculations, a valid installation and license are also required on the remote machine.

Both MATLAB and UQLAB are generally optional to create a DISPATCHER object. UQLINK MODEL evaluations do not require either. Similarly, `uq_map` computations can still be dispatched without UQLab, as long as they do not require any UQLab functionalities (see Appendix C).

A common practice in the organization of shared distributed computing resources (such as HPC clusters) is to have an *environment modules system* in place for managing the user-specific environment variables commonly associated with various toolsets. Consequently, while many common tools such as Matlab or MPI may have been installed on the remote machines, they will not be available to the user without the user explicitly *loading* these tools to their own environment. In an HPC cluster, the cluster is typically organized into *login* nodes and *compute* nodes. The user logs in to a login node to set up the computing environment and submit a computational job. The job itself will automatically be distributed to the compute node (or nodes) to be executed.

If this is the case, then the variables `EnvSetup` and `PrevCommands` must be specified accordingly. The commands specified inside `EnvSetup` will be executed once on the login node before the job is submitted (and distributed), while the commands specified inside `PrevCommands` will be executed on each compute node used to carry out the submitted job.

For example, an implementation of MPI and a particular version of Matlab can be loaded as follows:

```
EnvSetup = {'module load open_mpi'};
PrevCommands = {'module load matlab/R2015b'};
```

The command to load an MPI module on the remote machine is specified in `EnvSetup` because the associated command is used only once before the job is submitted. The command to load a version of Matlab, on the other hand, is specified in `PrevCommands` because the associated command (*i.e.*, `matlab`) is used on each of the compute nodes to which the job is distributed.

`EnvSetup` and `PrevCommands` are specified as a cell array to accommodate multiple commands.

Note: The particular environment module system and the available software are specific from an HPC cluster to another. Consult the documentation or the system administrator for support.

The last variable in the basic template specifies the job scheduler available on the remote machine. The Dispatcher module currently supports several scheduler out-of-the-box:

- None (no scheduler specified, direct execution)
- Slurm (`https://slurm.schedmd.com/documentation.html`)
- LSF (`https://www.ibm.com/support/knowledgecenter/en/SSWRJV_10.1.0/lsf_welcome/lsf_kc_ss.html`)
- PBS/TORQUE (`https://www.openpbs.org/`, `https://hpc-wiki.info/hpc/Torque`)

If the scheduler is, for instance, SLURM, then it can be specified in the profile file as follows:

```
Scheduler = 'slurm';
```

Note: If no scheduler is used on the remote machine (e.g. when using a remote workstation), then set the value of `Scheduler` to `'none'`.

Note that the vast majority of HPC clusters require the use of a scheduler. Please consult the documentation of your HPC resource or a system administration to identify the correct scheduler for your case.

If a scheduler is not supported by UQLab, users may specify the important settings associated with a custom scheduler. These settings are further explained in the next section.

B.2 Advanced template

An advanced profile template is available in:
`Profiles/HPC/profile_file_template_advanced.m`
inside the main UQLab installation folder. This template showcases more complex scheduler configurations in-depth.

In particular, it completely defines an entirely new custom `Scheduler`:

```
Scheduler = 'custom';
```

Because of the general nature of a custom scheduler, no defaults are assumed, and all the settings need to be specified by the user by defining the `SchedulerVars` structure.

Note: Users may also specify every value in `SchedulerVars` for the supported schedulers, if a more fine-tuned configuration is desired. In this case, any unspecified variables use their default values (see Table 5).

The first group of fields in `SchedulerVars` is related to creating a job script, which includes the syntax for the resources specification and standard output and error redirection.

For example, the LSF scheduler used the following entries as its default values:

```
SchedulerVars.Pragma = '#BSUB';
SchedulerVars.JobNameOption = '-J %s';
SchedulerVars.StdOutFileOption = '-oo %s';
SchedulerVars.StdErrFileOption = '-eo %s';
SchedulerVars.WallTimeOption = '-W %d';
SchedulerVars.NodesOption = '';
SchedulerVars.CPUsOption = '-n %d';
SchedulerVars.NodesCPUsOption = '';
SchedulerVars.CustomSettings = {};
```

Note that except for `.Pragma` and `.CustomSettings`, the above field values are given as a MATLAB string format specification. This specification is necessary for the DISPATCHER module to create the necessary submission strings on the remote resources. For instance, in `.JobNameOption`, `'%s'` will be replaced with a char value (*i.e.*, JOB name) by the DISPATCHER module when the job script is created, etc.

The second group of fields in `SchedulerVars` is related to the commands used to submit and cancel a job on the remote machine from the command line interface. For example, the TORQUE scheduler uses the following entries:

```
SchedulerVars.SubmitCommand = 'qsub';
SchedulerVars.CancelCommand = 'qdel';
```

These commands must be available on the login node of the remote system.

The field `.SubmitOutputPattern` stores a regular expression used to parse the output of the job submission command. Submitting a job script to a scheduler typically produces a string that contains the submitted job ID. This ID is necessary to keep track of the job as well as to manipulate/cancel it.

As an example, submitting a job script to SLURM will produce a string (by default): `'Submitted batch job <jobID>'`, where `<jobID>` is a numerical identifier (ID) of the submitted job.

Therefore, the following regular expression can be used by the DISPATCHER module to parse the submission job number of the SLURM scheduler:

```
SchedulerVars.SubmitOutputPattern = '[0-9]+';
```

The DISPATCHER module uses the expression above to extract the job ID from such a string.

The value of the field `.WorkingDirectory` is the environment variable provided by the scheduler that stores a remote job location from which the job is submitted.

As an example, the entry used by the LSF scheduler is as follows:

```
SchedulerVars.WorkingDirectory = '$LS_SUBCWD';
```

Finally, the field `.NodeNo` contains the environment variable that stores the compute node number in which the job is being executed[1]. The SLURM scheduler, for instance, uses the following entry:

```
SchedulerVars.NodeNo = '$SLURM_NODEID';
```

The above fields are typical of all common job schedulers employed in HPC resources. Proper configuration values are normally found in the scheduler documentation.

[1] for a parallel job, the job may be distributed to multiple nodes.

Appendix C

Dispatching uq_map

Note: This appendix only details the particularities of dispatched uq_map. Unless stated otherwise, all options for the local execution of uq_map work as is. The reader is referred to the uq_map chapter in UQLIB user manual for a detailed user guide on uq_map.

C.1 Syntax

```
[OUTPUT_1,...,OUTPUT_NOUT] = uq_map(FUN, INPUTS, DISPATCHEROBJ)
uq_map(FUN, INPUTS, DISPATCHEROBJ, 'ExecMode', 'async')
uq_map(..., NAME, VALUE)
```

`[OUTPUT_1,...,OUTPUT_NOUT] = uq_map(FUN, INPUTS, DISPATCHEROBJ)` maps a sequence `INPUTS` to another sequence by evaluating a function handle `FUN` on each elements of `INPUTS`. All the evaluations are carried out on the remote machine using the DISPATCHER object `DISPATCHEROBJ`. Calling `uq_map` creates a JOB in `DISPATCHEROBJ`. The remote execution is synchronized with the local UQLAB session; that is, the local session waits for the remote JOB to finish and the results are available. The results on the remote machine are then automatically fetched back to the local session. Depending on `FUN`, multiple output arguments may be supported.

`uq_map(FUN, INPUTS, DISPATCHEROBJ, 'ExecMode', 'async')` switches to the non-synchronized execution mode. That is, the local UQLAB session does not wait for the remote JOB to finish; once the JOB is successfully dispatched to the remote machine, control is given back to the local session. If the results are assigned to a variable, an empty vector is returned. The results of the execution need to be fetched manually by the user with the `uq_fetchResults` command (see Section 2.3.3).

`uq_map(..., NAME, VALUE)` maps a sequence `INPUTS` to another sequence by evaluating `FUN` on each element of `INPUTS` on the remote machine with additional (optional) `NAME`/`VALUE` argument pairs (see Section C.3).

C.2 Description

In the subsequent sections, basic and advanced uses of a dispatched `uq_map` are presented assuming that a DISPATCHER object has been set up properly and named `myDispatcher` (see Section 2.2 for details).

C.2.1 Basic usage

A single function evaluation can be dispatched to the remote machine by calling `uq_map` with a DISPATCHER object. A simple example of such use case is dispatching a MATLAB built-in function (`sum`) with a vector as input and followed by a DISPATCHER object:

```
Y = uq_map(@sum, {linspace(1,100)}, myDispatcher)
```

A JOB associated with this evaluation is created on, submitted to, and executed on the remote machine. By default, the remote execution is synchronized with the local UQLAB session; this means the local UQLAB session is blocked after the evaluation is successfully dispatched and the session will remain blocked until the remote JOB is finished. Once successfully finished, the results are automatically fetched back to the local session:

```
Checking the status of the remote execution...
Checking the status of the remote execution...
Job Status: 'complete' reached.
Y =

  1x1 cell array

    {[5050]}
```

The result is identical to the local evaluation of `uq_map`:

```
uq_map(@sum,{linspace(1,100)})
```

```
ans =

  1x1 cell array

    {[5050]}
```

C.2.2 Synchronized vs. non-synchronized execution mode

By default, dispatched `uq_map` evaluations are executed in synchronized execution mode, to preserve the user experience of local computations. For non-synchronized execution, the named argument `'ExecMode'` followed by the char value `'async'` can be used. In this mode, the local MATLAB session is not blocked and users can continue working in the interactive session while the function evaluation is executed on the remote machine. Consequently, the results will not be available when the function returns to the interactive session (Figure 5).

For example, the following uq_map call dispatches and evaluates the sum function asynchronously:

```
Y = uq_map(@sum, {linspace(1,100)}, myDispatcher,...
    'ExecMode', 'async')
```

By default, when the output is assigned to a variable and the dispatched computation is executed asynchronously, an empty array is returned:

```
Y =

         []
```

Users can wait for the remote execution to finish by calling the uq_waitForJob function on the DISPATCHER object:

```
uq_waitForJob(myDispatcher)
```

which may display the following:

```
Checking the status of the remote execution...
Job Status: 'complete' reached.
```

The results can only be fetched back to the local client if the JOB has finished successfully (*i.e.*, it reaches a 'complete' status). To check the status of the Job, use the uq_getStatus function on the DISPATCHER object:

```
uq_getStatus(myDispatcher)
```

```
ans =

    'complete'
```

To fetch the results of the remote evaluation, use the uq_fetchResults function on the DISPATCHER object:

```
Y = uq_fetchResults(myDispatcher)
```

```
Y =

  1x1 cell array

    {[5050]}
```

C.2.3 Multiple JOB objects

Multiple JOBs can be associated with a single DISPATCHER object. Every time uq_map is called with a DISPATCHER object, a JOB is created and associated with the DISPATCHER object. The following consecutive calls create four dispatched uq_map evaluations, and consequently, four remote JOBS:

```
uq_map(@sum, {linspace(1,100)}, myDispatcher)
uq_map(@sum, {linspace(-1,100)}, myDispatcher)
uq_map(@sum, {linspace(1,10)}, myDispatcher)
uq_map(@sum, {linspace(-1,10)}, myDispatcher)
```

If the computation is executed in the non-synchronized mode, a new uq_map call can be made regardless on the status of the other previously submitted JOBS.

To see an overview of all the JOBS associated with a DISPATCHER object, use the uq_listJobs on the DISPATCHER object:

```
uq_listJobs(myDispatcher)
```

results in something similar to the following:

```
Dispatcher Unit: Dispatcher 2

No.  Job ID   Status     Tag                                        Finish Date Time...
----------------------------------------------------------------------------------------...
 1   1871     complete   uq_map of <sum> on <20-Aug-2020 14:02:34>  08/20/20 12:02:4...
 2   1872     complete   uq_map of <sum> on <20-Aug-2020 14:02:39>  08/20/20 12:02:4...
 3   1873     submitted  uq_map of <sum> on <20-Aug-2020 14:02:45>
 4   1874     running    uq_map of <sum> on <20-Aug-2020 14:02:51>
```

Because multiple JOBS can be associated with a single DISPATCHER object, all the functions to interact with a JOB already mentioned such as uq_print, uq_waitForJob, uq_getStatus, and uq_fetchResults accept as their second argument the index for selecting a JOB. For instance, calling:

```
uq_fetchResults(myDispatcher,2)
```

fetches the results of the JOB indexed with 2 in the DISPATCHER object.

Every time a JOB is created, the new JOB is appended to the Jobs property (an object array) of the DISPATCHER object. That is, the last created JOB is located at the end of the Jobs property. By default, without specifying the index, the function will apply to the last created JOB in the DISPATCHER object.

C.2.4 Parallel execution

uq_map accepts an input sequence for the mapping function. In this case, the mapping function is evaluated on each element of the input sequence one at a time (see the chapter on uq_map in the UQLIB user manual).

If more than one remote process is requested in the DISPATCHER object and there is more than one element in the input sequence, the dispatched uq_map will automatically be evaluated in parallel. That is, the input sequence will be divided into as many slices as the requested remote processes. All the slices will be then evaluated in parallel on the remote machine.

To illustrate this point, dispatch the same sum function on multiple input vectors as follows:

```
myDispatcher.NumProcs = 2;
inVectors = {linspace(1,100); linspace(-1,100);...
  linspace(1,10); linspace(-1,10)};
uq_map(@sum, inVectors, myDispatcher)
```

In the above call, there are four elements in the input sequence. Because the number of processes specified in the DISPATCHER (NumProcs property) is two, there will be two parallel uq_map evaluations, each of which operates on two elements of the input sequence.

Once the JOB has finished execution, the results are shown below:

```
Checking the status of the remote execution...
Job Status: 'complete' reached.
ans =

  4x1 cell array

    {[5050]}
    {[4950]}
    {[ 550]}
    {[ 450]}
```

Notice that these results are exactly the same as the following local and serial evaluation of uq_map:

```
uq_map(@sum, {linspace(1,100); linspace(-1,100);...
  linspace(1,10); linspace(-1,10)})
```

```
ans =

  4x1 cell array

    {[5050]}
    {[4950]}
    {[ 550]}
    {[ 450]}
```

In other words, the parallel evaluation in the remote machine is transparent to users; there is no difference in the end results between the local and dispatched evaluation of uq_map.

C.2.5 Multiple output arguments

As in the case of local execution, dispatching uq_map calculations also supports functions with multiple output arguments.

For instance, the built-in MATLAB function svd can return three output arguments: the matrices U, S, and V. The following illustrates the svd computation on two different matrices using dispatched uq_map in which the first two of the three outputs are requested:

```
A = [1 0 1; -1 -2 0; 0 1 -1];
B = [1 2; 3 4; 5 6; 7 8];
[U,S] = uq_map(@svd, {A B}, myDispatcher)
```

```
Checking the status of the remote execution...
Job Status: 'complete' reached.

U =

  1x2 cell array

    {3x3 double}    {4x4 double}

S =

  1x2 cell array

    {3x3 double}    {4x2 double}
```

Each output argument contains the output of `svd` on each element of the inputs.

When the outputs are not explicitly assigned to multiple variables (*e.g.*, when non-synchronized execution is chosen), the additional argument `'NumOfOutArgs'` is required to inform the DISPATCHER about the expected number of output arguments. If this argument is not specified, only a single output (the first) will be stored in the DISPATCHER results. For instance, using the named argument for the above `svd` computation:

```
uq_map(@svd, {A B}, myDispatcher,...
    'ExecMode', 'async',...
    'NumOfOutArgs', 2)
```

Once the JOB has successfully finished, the results with multiple output arguments can be fetched (and assigned to variables) as follows:

```
[U,S] = uq_fetchResults(myDispatcher)
```

```
U =

  1x2 cell array

    {3x3 double}    {4x4 double}

S =

  1x2 cell array

    {3x3 double}    {4x2 double}
```

By default, if the number of output arguments are not specified (either via variables assignment or the named argument `'NumOfOutArgs'`), the first output argument is always requested.

A partial selection of output arguments is also supported. For instance, the following calls:

```
[~,S] = uq_fetchResults(myDispatcher)
```

or

```
[U,~,V] = uq_map(@svd, {A B}, myDispatcher)
```

are supported.

Note: Note that the ability to fetch multiple outputs from dispatched `uq_map` depends on the mapping function and how the evaluation is initially dispatched. For instance, if `uq_map` is dispatched with only a single output argument, then the results cannot be fetched for more than one output, regardless of whether the mapping function supports multiple output arguments or not.

C.2.6 Dispatching user-defined functions

The evaluation of user-defined functions may also be dispatched using `uq_map`. For instance, suppose the user-defined function is the following:

```
function Y = xsinx(X)
  Y = X .* sin(X)
end
```

The function is saved as `'xsinx.m'` and assumed to be available in the local MATLAB search path.

The function is evaluated on four values of `x` using `uq_map` as follows:

```
uq_map(@xsinx, [-pi -0.5*pi 0.5*pi pi], myDispatcher)
```

The `m`-file of the user-defined function will automatically be included in the dispatch package sent to the remote machine by the DISPATCHER object. However, this automatic inclusion is limited to a simple function without any additional dependencies on other user-defined functions. If there is a dependency on a file or a set of files (e.g. when calling a third party MATLAB-based software), these files must be explicitly made available on the remote machine. Similarly, if a user-defined function is also wrapped inside another anonymous function, then the user-defined function must also be explicitly made available on the remote machine. In other words, all the files needed for the evaluation of a task must be available on the remote machine.

Section C.2.6.1 discusses different approaches to achieve this directly through the UQLAB DISPATCHER.

C.2.6.1 Attaching files and adding remote paths

In case the mapping function in `uq_map` depends on other user-defined files (be they data files or other function files), these files must be explicitly made available on the remote machine. There are several ways to achieve this:

- Specify the required files or folders in the call to `uq_map` using the named argument `'AttachedFiles'`, followed by a cell array that contains the list of attached files and folders. The cell array may contain an arbitrary number of file and folder specifications. Every time `uq_map` is called, the specified files and folders will be copied to the remote machine. This is a recommended approach for relatively small helper functions or infrequently used functions.

  Note: To avoid unexpected behavior due to collisions with names already in the MATLAB search path, it is recommended to specify the absolute paths for the files and folders.

- Manually copy all the required files to a location in the remote machine and specify the paths to these files and folders in the call to `uq_map` using the named arguments `'AddToPath'` or `'AddTreeToPath'`. In this case, the user is responsible to prepare the remote machine for execution prior to submitting the `uq_map` command.

  Both `'AddToPath'` and `'AddTreeToPath'` are cell arrays that may contain an arbitrary number of path specifications. The paths must be accessible by the remote MATLAB session. This is a recommended approach for frequently used files or a large set of files.

  The difference between these two options is that in the case of `'AddTreeToPath'` all the sub-directories within each given path will be included in the remote MATLAB search path.

All the files attached and paths added using the above schemes are specific to the JOB created from a particular call to the `uq_map` with the named arguments.

Note: Alternatively to the second option above, the corresponding DISPATCHER object properties `'AddToPath'` and `'AddTreeToPath'` may also be used and specified the same way. However, unlike using the named arguments, the paths specified to these properties will apply to all JOBs created by the DISPATCHER, regardless of the dispatcher-aware command. In other words, the properties act as global settings for the subsequent JOBs.

As an example, consider the following use case. The mapping function file `'myFunction.m'` reads:

```
function Y = myFunction(idx)

X = dlmread(sprintf('myData%04d.csv'),idx);
Xp = preprocess(X);
Y1 = stepA(Xp);
Y2 = stepB(Xp);
```

```
Y = Y1 + Y2;

end
```

The function takes as its input an index that indicates the data file to read before processing it in multiple steps using several different user-defined functions (each of which is in its own file). Therefore, `myFunction` depends on several other functions and on a set of data files located in a directory structure on the client:

```
/path/to/files
├─preprocessX.m
├─myFunctionLib
│ ├─stepA.m
│ └─stepB.m
└─myDataFiles
  ├─myData0001.csv
  ├─myData0002.csv
  ├─...
  └─myData1000.csv
```

To dispatch the function evaluation on a set of indices and resolve all the dependencies, `uq_map` is called with the named argument `'AttachedFiles'` as follows:

```
    ListOfFiles = fullfile('path/to/files',...
    {'preprocessX.m', 'myFunctionsLib', 'myDataFiles'});
    uq_map(@myFunction, (1:1000)', myDispatcher,...
    'AttachedFiles', ListOfFiles)
```

Specifying the absolute paths to these files and folders is not mandatory but recommended to avoid unexpected behaviors due to collision with names already in the MATLAB path.

In order to minimize network traffic, if the above `uq_map` is to be called frequently or the data files are particularly large or numerous, then it is recommended instead to copy *in advance* all these files to a location in the remote machine (*e.g.*, `'/home/jdoubt/data'`), and then call `uq_map` with the named argument `'AddTreePath'` instead.

For example:

```
uq_map(@myFunction, (1:1000)', myDispatcher,...
  'AddTreeToPath', {'/home/jdoubt/data'})
```

Note once more that the paths specified in both `'AddToPath'` and `'AddTreeToPath'` correspond to the paths on the remote machine.

C.2.7 Dealing with large input sequences

Every time `uq_map` is called, a dispatch package is created and sent to the remote machine. Included in this dispatch package is the input sequence. If the input sequence passed to uq_map is large, the corresponding file may create a bottleneck both in during generation in the local UQLAB session and later during its transfer over the network. Moreover, local client machines typically have more limited memory compared to remote machines; this, in turn, may prohibit the generation of particularly large sequences in the first place.

To circumvent this problem, uq_map allows the instruction to generate the sequence of inputs to be put inside a function (referred to as a *sequence generator*). This function will then be executed in the remote machine, and the sequence is made available for the remote uq_map.

The sequence generator is a MATLAB function having the following signature:

```
function mySequence = mySequenceGenerator(varargin)
```

where the argument varargin is can be used to specify an optional set of parameters to be passed to the sequence generator from one uq_map call to another. The output of this function can be any type of sequence supported by uq_map, including cell arrays, structure arrays, vectors, and matrices (see the UQLIB user manual for more detail).

If a sequence generator is used to generate the input sequence on the remote machine, the named argument 'InputSize' followed a vector indicating the size of the input must be specified. This information is important for splitting parallel tasks as well as reconstructing the output and it cannot be inferred by the client when uq_map is called.

For example, given the following sequence generator function:

```
function mySequence = mySequenceGenerator(varargin)
  mySequence = randn(1e4,1e5);
end
```

the sum function is evaluated on each rows of the matrix with the following uq_map call:

```
uq_map(@sum, @mySequenceGenerator, myDispatcher,...
  'InputSize', [1e4 1e5], 'MatrixMapping', 'ByRows')
```

By default, the generator function file is automatically included in the dispatch package.

To change the behavior of a sequence generator from one uq_map call to another, a set of parameters can be passed using the named argument 'SeqGenParameters' followed by a cell array that contains the parameters. For example, in the following sequence generator function:

```
function mySequence = mySequenceGenerator(varargin)
  rng(varargin{3},'twister')
  mySequence = varargin{1} + varargin{2} * randn(1e4,1e5);
end
```

there are three parameters: the mean and the standard deviation of a normal distribution as well as the seed number for the random number generator. The following uq_map call evaluates the column-wise mean and standard deviation of a sample generated from a normal distribution with mean 1 and standard deviation 2:

```
uq_map(@(X) [mean(X) std(X)], @mySequenceGenerator, myDispatcher,...
  'MatrixMapping', 'ByColumns',...
  'InputSize', [1e4 1e5],...
  'SeqGenParameters', {1, 2, 100})
```

C.2.8 Dispatching UQLab functions and objects

UQLAB functions may be passed as the mapping function to uq_map, provided that a licensed UQLAB installation available on the remote machine, and the DISPATCHER object has been set up properly (see Section 2.2). To enable UQLAB on the remote machine, the named argument 'UQLab' followed by a logical value true is used (by default, the value is false).

To illustrate this feature, three UQLAB MODEL objects, based on a set of simple strings:

```
ModelOpts(1).mString = 'X * sin(X)';
ModelOpts(2).mString = 'X * cos(X)';
ModelOpts(3).mString = 'X * tan(X)';
```

are created in the remote machine using the following uq_map call:

```
uq_map(@uq_createModel, ModelOpts, myDispatcher, 'UQLab', true)
```

After the Job is finished, fetching the results yields:

```
uq_fetchResults(myDispatcher)
```

```
ans =

  1x3 cell array

    {1x1 uq_model}    {1x1 uq_model}    {1x1 uq_model}
```

As can be seen, three MODEL objects have been created on the remote machine and fetched back to the local client.

The current UQLab session in the local MATLAB session can also be saved and then loaded in the remote machine. This is achieved by using the named argument 'SaveUQLab' followed by a logical value true (by default, the value is false). This construction is useful to make all the current UQLAB objects accessible in the remote machine.

As an example, a MODEL object is first created in the local MATLAB session:

```
ModelOpts.mString = 'X * sin(X)';
myModel = uq_createModel(ModelOpts);
```

then dispatched and evaluated on a set of input values in the remote machine using the following uq_map call:

```
uq_map(@uq_evalModel, (-pi:0.1:pi)', myDispatcher,...
    'UQLab', true, 'SaveUQLabSession', true)
```

If the MODEL object is not explicitly specified, then by default, `uq_evalModel` evaluates the *current* (*i.e.*, the last created) UQLAB MODEL object. This implicit syntax is possible on the remote machine only if the MODEL object itself is made available on the remote machine.

C.2.9 Handling errors

By default, any error from evaluating a mapping function on one of the inputs is automatically handled by returning a `NaN` value in the cell element of the corresponding output. For diagnostic purposes, 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 cause `uq_map` to fail, therefore killing the entire JOB.

A custom user-defined error handler may also be specified; the reader is referred to the `uq_map` chapter of the UQLIB user manual for details. In this case, the error handler must be saved in a separate `m`-file, which will automatically be included in the dispatch package. As noted in Section C.2.6.1, if the error handler depends on some other files, then these files must be manually available for the remote MATLAB session, either by attaching them or adding their location on the remote machine to the remote MATLAB search path.

C.2.10 Naming and tagging uq_map JOBS

Every JOB automatically created when `uq_map` is called with a DISPATCHER object is assigned a standardized name. By default, the name of a JOB is the timestamp at which the JOB is created up to the milliseconds with the following format: `<ddMMMyyyy>_at_<HHmmssSS>`[1].

A different name can be explicitly assigned by passing the named argument `'Name'` followed by a char array:

```
Y = uq_map(@sum, {linspace(1,100)}, myDispatcher, 'Name', 'myNewJob')
```

Additionally, one can tag a JOB, *e.g.*, with a short text description of the JOB. This description is to help users identify the different JOBs that have been submitted. Tags are printed when the JOBs associated with a DISPATCHER object is listed (see Section C.2.3). By default, the tag of a JOB created by `uq_map` has the following template:

```
'uq_map of <funName> on <dd-mm-yyyy HH:mm:ss>')
```

where `<funName>` and `<dd-mm-yyyy HH:mm:ss>` are placeholders for the name of the mapping function and the current date and time (when the particular `uq_map` is called), respectively.

Users may change this default for a specific JOB by passing the named argument `'Tag'`

[1]in the standard MATLAB Date format

followed by a char array. For example:

```
Y = uq_map(@sum, {linspace(1,100)}, myDispatcher,...
    'Tag', 'My important remote computation')
```

The tag can be edited after it has been created by directly editing the `Tag` property of the appropriate JOB object.

For instance, to change the tag of the last created JOB, type:

```
myDispatcher.Jobs(end).Tag = 'Not so important remote computation'
```

C.2.11 Dispatching Linux system commands

A special use case of `uq_map` coupled with a DISPATCHER object is to evaluate a system command on the remote machine. Because remote machines are assumed to be running a Linux operating system, the system command must be a Linux system command. When a system command is dispatched, the Bash shell interpreter is required on the remote machine. On the other hand, no installation of MATLAB is required.

System commands are specified by assigning a `char` string as the mapping function, instead of a function handle. Composing such a system command follows the syntax explained in the `uq_map` chapter of UQLIB user manual. Furthermore, only data types that can be represented as a char array can be used as the input sequence element.

As an example, the following `uq_map` call dispatches and evaluates the system command to compute the sum of four pairs of numbers (note that the program `bc` is assumed to be available on the remote machine):

```
uq_map('echo {1}+{2} | bc', {{1 2}; {3 4}; {5 6}; {8 7}},...
  myDispatcher)
```

In the case of a dispatched system command evaluation, there are no results to fetch. However, the output streams (standard output and standard error) are automatically fetched. These are available in the `OutputStreams` property of the corresponding JOB object. For example:

```
myDispatcher.Jobs(end).OutputStreams
```

```
ans =

  struct with fields:

      SubmitStdOut: {'Submitted batch job 1896'}
         JobStdOut: {1x3 cell}
         JobStdErr: {''}
     ProcessStdErr: {2x1 cell}
        TaskStdOut: {4x1 cell}
        TaskStdErr: {4x1 cell}
```

```
    TaskExitStatus: {4x1 cell}
```

The results of the summation is stored in the standard output of the command execution. For instance the third summation (*i.e.*, $5 + 6$) is:

```
myDispatcher.Jobs(end).OutputStreams.TaskStdOut{3}
```

```
ans =

  1x1 cell array

    {'11'}
```

C.2.12 Fetching the output streams of the remote execution

The output streams related to the remote execution (*i.e.*, the standard output and standard error) can be fetched back to the local client machine using the function `uq_fetchOutputStreams` on a DISPATCHER object. This often useful for diagnostic purposes, especially if the JOB fails.

By default, `uq_fetchOutputStreams` fetches and returns the output streams of the last created JOB on the DISPATCHER object. The output streams are returned as a structure. For example:

```
myOutputStream = uq_fetchOutputStreams(myDispatcher)
```

```
myOutputStream =

  struct with fields:

      SubmitStdOut: {'Submitted batch job 1907'}
         JobStdOut: {1x18 cell}
         JobStdErr: {''}
     ProcessStdErr: {2x1 cell}
        TaskStdOut: {}
        TaskStdErr: {}
    TaskExitStatus: {}
```

A particular field of the output stream structure can be printed using the `uq_printStream` function on an output stream structure followed by the selected fieldname. For instance, to print the standard output of the whole remote job, use:

```
uq_printStream(myOutputStream,'JobStdOut')
```

Below is an example of a standard output:

```
Running on host 123.456
Time is Fri 21 Aug 2020 02:50:28 PM CEST
Directory is /home/jdoubt/temp/21Aug2020_at_14502392
                            < M A T L A B (R) >
                  Copyright 1984-2019 The MathWorks, Inc.
```

```
                    R2019b (9.7.0.1190202) 64-bit (glnxa64)
                                August 21, 2019
                              < M A T L A B (R) >
                    Copyright 1984-2019 The MathWorks, Inc.
                    R2019b (9.7.0.1190202) 64-bit (glnxa64)
                                August 21, 2019

To get started, type doc.
For product information, visit www.mathworks.com.


To get started, type doc.
For product information, visit www.mathworks.com.
```

By default, calling `uq_fetchOutputStreams` does not update the JOB object inside the DISPATCHER object. To update the `OutputStreams` property of the JOB object, the named argument `'UpdateDispatcher'` followed by a logical value `true` can be passed to `uq_fetchOutputStreams`.

Note: If the remote execution fails, then by default, the output streams are automatically fetched and available in the `OutputStreams` property of the JOB object.

C.2.13 Current limitations

While most features and options of `uq_map` work out of the box when dispatched, it is worthwhile to note the following limitations:

- If a user-defined function is used as the mapping function in `uq_map`, then an automatic attachment of the function file in the dispatch package only works for a simple function without any dependencies on other user-defined files or data files. If the mapping function depends on other function files or data files, these files must be manually made available for the remote MATLAB session. The different approaches to achieve this are outlined in Section C.2.6.1.
- Even for a simple function, if the function is wrapped inside an anonymous function, the automatic attachment would not be able to include the relevant function file. In this case, the file must also be manually made available for the remote MATLAB session.

C.3 Input

Below is a brief reference list of the dispatcher-specific options of `uq_map`. For a full reference list, please refer to UQLIB user manual.

Table 20: `uq_map(FUN, INPUTS, DISPATCHEROBJ, NAME, VALUE)`

●	FUN	Function handle or char array	Mapping function, used to evaluate (*i.e.*, *map*) each element of INPUTS. See the chapter on uq_map in UQLIB user manual for some remarks on the supported types of function.
●	INPUTS	Cell array, structure array, matrix, or function handle	Sequence whose elements are passed as inputs into FUN. See Section C.2.7 for details if a function handle is passed as the sequence.
●	DISPATCHEROBJ	DISPATCHER object	DISPATCHER object to dispatch the evaluation to a remote machine.
☐	NAME, VALUE	name-value pair (See Table 21)	Additional options as name-value pairs.

Table 21: 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 the chapter on uq_map in UQLIB user manual for details and a usage example.
'NumOfOutArgs'	Scalar integer default: 1	Number of output arguments to get from each evaluation of FUN, if FUN is a function that returns multiple outputs. See Section C.2.5 for details and a usage example.
'ExpandCell'	logical default: false	Flag to expand the content of a cell into a comma-separated list. This is useful if the contents of a cell element are to be pass as a list of arguments to FUN. See the chapter on uq_map in UQLIB user manual for details and a usage example.

Continued on next page

Table 21–continued from previous page

'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 the chapter on uq_map in UQLIB user manual for details and a usage example.
'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 the chapter on uq_map in UQLIB user manual for details and a usage example.
'ExecMode'	String default: 'sync'	Execution mode of the dispatched computation. In the non-synchronized execution mode ('async'), if the call to uq_map is assigned to a variable, then an empty vector is returned. On the other hand, in the synchronized mode ('sync'), the local UQLAB session waits for the remote execution to finish and, once finished, the results are automatically fetched back to the local session. See Section C.2.2 for details.

Continued on next page

Table 21–continued from previous page

`'UQLab'`	logical default: `false`	Flag to load UQLab in the remote Matlab session. This requires a licensed UQLab to be made available in the remote machine and the Dispatcher object set up properly with the corresponding path. See Section C.2.8 for details.
`'SaveUQLabSession'`	logical default: `false`	Flag to save the current local UQLab session and load it on the remote Matlab session. See Section C.2.8 for details.
`'AttachedFiles'`	String or cell array default: `{}`	List of files and folders on the local client to be copied and made available on the remote machine. See Section C.2.6.1 for details.
`'AddToPath'`	String or cell array default: `{}`	List of paths on the remote machine to be added to the PATH of the remote execution environment (*e.g.*, Matlab search path). See Section C.2.6.1 for details.
`'AddTreeToPath'`	String or cell array default: `{}`	List of paths (including subdirectories) in the remote machine to be added to the PATH of the remote execution environment (*e.g.*, Matlab search path). See Section C.2.6.1 for details.
`'Tag'`	String	Descriptive text for the `uq_map` Job. See Section C.2.10 for details.
`'Name'`	String default: `uq_createUniqueID()`	Name of the `uq_map` Job. See Section C.2.10 for details.
`'FetchStreams'`	Logical default: false	Flag to bring the captured standard error and output back to the local Matlab session and stored in the corresponding Job object. If the Job on the remote fails, the output stream will be automatically captured. See Section C.2.12 for details.
`'AutoSubmit'`	Logical default: true	Flag to automatically submit the remote Job associated with the call to `uq_map`.

C.4 Output

Table 22: [OUTPUT_1,...,OUTPUT_NOUT] = uq_map(...)

OUTPUT	Cell array	Results of evaluating FUN on each element of INPUTS. Using the non-synchronized execution mode, an empty vector is returned after successfully dispatching the JOB; once finished, the results of the remote computation must be fetched using uq_fetchResults. 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.

C.5 Notes

- When uq_map is dispatched using a DISPATCHER object, a JOB object will automatically be created as a side effect and associated with the DISPATCHER object. The newly created JOB object will be appended to the Jobs property of the DISPATCHER object.

Bibliography

Ishigami, T. and T. Homma (1990). An importance quantification technique in uncertainty analysis for computer models. In *Proc. ISUMA'90, First Int. Symp. Unc. Mod. An*, pp. 398–403. University of Maryland. 13

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. 29

Moustapha, M., C. Lataniotis, P. Wiederkehr, D. Wicaksono, S. Marelli, and B. Sudret (2021). UQLab user manual – UQLib. Technical report, Chair of Risk, Safety & Uncertainty Quantification, ETH Zurich. Report # UQLab-V1.4-201. 7

Moustapha, M., S. Marelli, and B. Sudret (2021). UQLab user manual – the UQLink module. Technical report, Chair of Risk, Safety & Uncertainty Quantification, ETH Zurich. Report # UQLab-V1.4-110. 26