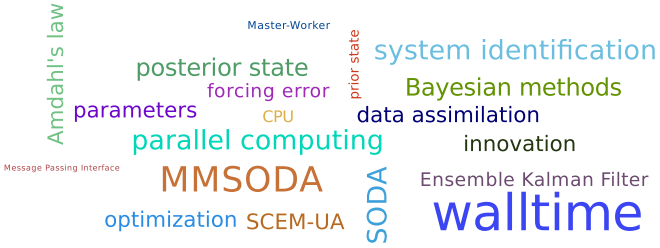
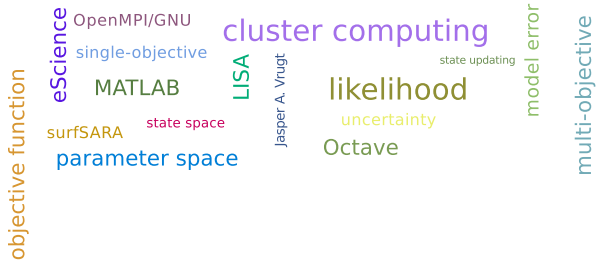


The MMSODA Toolbox for MATLAB



University of Amsterdam

April 2013

Author: Jurriaan H. Spaaks

Contents

Chapter 1

Introduction

This document is a manual for learning how to use the LISA cluster computer hosted at SURFsara (https://www.surfsara.nl/systems/lisa) for solving parameter tuning and system identification problems. The manual covers the basic organization of the cluster's hardware, and provides the user with some simple commands you'll need for manipulating the Linux system that LISA is running. The manual further introduces the MMSODA parameter tuning and system identification software that can be run on LISA. While going through this manual, you'll find that most of it is aimed at Windows users, but occasionally there will be brief instructions for Linux users as well (usually in the footnotes). It is assumed that you are proficient in using MATLAB, and that you have a working MATLAB installation on your local machine. Furthermore, it is assumed that you have a basic understanding of some of the general concepts used in optimization.

Chapter 2

Getting started

Effective usage of the LISA system requires that you:

1. have a basic understanding of how the LISA system is organized;
2. know how to copy files from your local machine to LISA and back using SFTP;
3. know how to establish an SSH connection, linking your local machine to the LISA system;
4. have a basic understanding of the Linux commands that are needed to tell the LISA system what it is you want to do.

In the remainder of this Chapter, we cover these items in relevant detail.

2.1 Cluster computing

A *cluster computer* is a system of interconnected computers that work together so that in many respects they can be viewed as a single system. Clusters usually consist of regular off-the-shelf computers such as those you may find in any computer retail shop (see Fig. 2.1). Cluster computers are especially suited for solving a certain class of computational problems, namely those problems that can be divided into smaller, independently solvable tasks. As a trivial example, finding the minimum value in a 2-D array of values (i.e. a map) is such a problem: the map can be split into two parts and sent to two separate machines. After both machines have found the minimum value in the part of the map that they were assigned, the 'global' minimum can be determined by simply comparing the two values. Dividing the task over two or more machines (or actually, *cores*) is called *parallelization*. Parallelization can greatly reduce the *walltime* of a computational task (i.e. the time between starting a task and knowing the solution).

Figure 2.1: Technicians at work on a real cluster computer. Photo from http://en.wikipedia.org/wiki/File:MEGWARE.CLIC.jpg.

2.2 The LISA cluster: hardware

The LISA cluster currently[1] consists of 624 normal computers, which are referred to as *nodes*. In terms of hardware, each node has a Central Processing Unit (CPU), disk storage, and memory. The nodes are interconnected using a Local Area Network (LAN). The LAN has low latency and high bandwidth (Gigabit or Infiniband), meaning that in terms of time it is cheap to send large files (because of the high bandwidth) as well as small files (because of the low latency). Storage space on each node varies a little bit, but at the time of writing is 70–220 GB per node. Most nodes have 8-core CPUs, although some have 12-cores or 16-cores. The clock frequency for individual cores is 1.80–2.26 GHz. In terms of memory, most nodes have 24 GB, but the 16-cores have 32 GB. Furthermore, the bandwidth between memory and CPU varies between 5.86–8.00 GT/s. Finally, all nodes run the Debian Linux AMD64 operating system.

[1] LISA's hardware is subject to regular upgrades. For concurrent information, see https://www.surfsara.nl/systems/lisa/description#System_configuration

2.3 Transferring files to and from the LISA cluster

This section documents the necessary steps for connecting to the LISA cluster from a local Windows machine. First, make sure that you have an account for accessing LISA[1]. In order to use the cluster effectively, you need to be able to copy files to and from your user directory on the cluster. The LISA cluster is set up such that it exclusively allows connections that are secure, such as Secure File Transfer Protocol (SFTP) connections or Secure Shell (SSH) connections. There are many programs that can establish secure connections, but we will use WinSCP[2].

▶ 1. Download WinSCP from http://sourceforge.net/projects/winscp/files/WinSCP/4.3.7/winscp437.zip/download[3,4]. Unzip into your local home directory or a USB drive. Double-click on 'WinSCP.exe' to start the WinSCP program. The program will prompt you for some input (see Fig. 2.2). Under 'Host name:', fill in 'lisa.surfsara.nl'. Make sure that the port number is set to '22'. Under 'User name:', fill in your LISA cluster username. You can leave the 'Password' field blank, the program will prompt you later. Make sure 'SFTP' is selected as the file protocol. Click on 'Save...' to store these settings if you like. Then press the 'Login' button to establish the SFTP connection to the LISA cluster. The program will throw a warning about the RSA key fingerprint file. The number shown in the warning dialog should be the same as the number posted at https://www.surfsara.nl/systems/shared/ssh (table at the bottom of the web page). If it is, press 'Yes' and then 'Continue' in the next dialog box. As a final step, fill in your password when prompted. You should now see two panes, one for the local machine on the left and one for the remote machine, i.e. the LISA cluster, on the right (see Fig. 2.3).

▶ 2. You should have received 'tutorial.zip' with this document. Unzip it into your (local) home directory or a USB drive.

▶ 3. In WinSCP, use the left pane to navigate to the directory where you unzipped the tutorial files. Copy the 'tutorial' folder to the remote directory by selecting it on the left and pressing the F5 button. The tutorial folder should now be present in the right pane as well.

[1] You can get an account by following the instructions from https://www.surfsara.nl/systems/lisa/account.

[2] If you elect not to use WinSCP, make sure that your alternative program can handle file permissions correctly when transferring between Windows and Linux. Also make sure that line endings are converted properly. Check http://en.wikipedia.org/wiki/Newline for an interesing read on the many ways in which end-of-line characters may cause you trouble. Finally, if you have a text file with Windows-style line endings (CR/LF) and you want to convert it to Linux-style line endings (LF), make sure to check out the free, open-source, Notepad++ editor at http://notepad-plus-plus.org/. This excellent program lets you convert between line endings in just a few mouse clicks.

[3] By the time you read this, there may be a more recent version available—just substitute '4.3.7' and '437' from the address with the version you want. For an overview of available versions, look here: http://sourceforge.net/projects/winscp/files/WinSCP/.

[4] Life is somewhat easier for Linux users. Most Linux distributions come with built-in support for secure connections. Just start up your regular file browser (e.g. PCManFM, Thunar, Nautilus, etc) and type in the address bar sftp://jspaaks@lisa.surfsara.nl/home/jspaaks. Don't forget to substitute your own username. This should automatically establish a connection between your machine and the remote system (i.e. the cluster). Fill in your credentials when prompted.

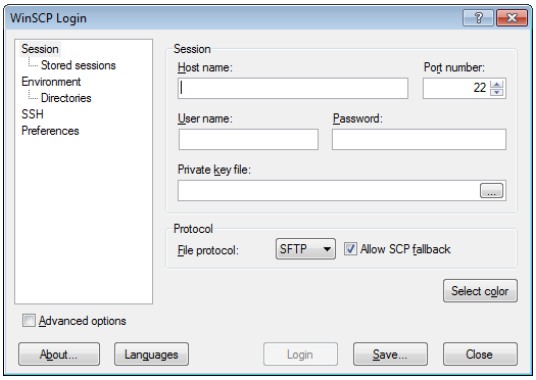


Figure 2.2: WinSCP session dialog box.

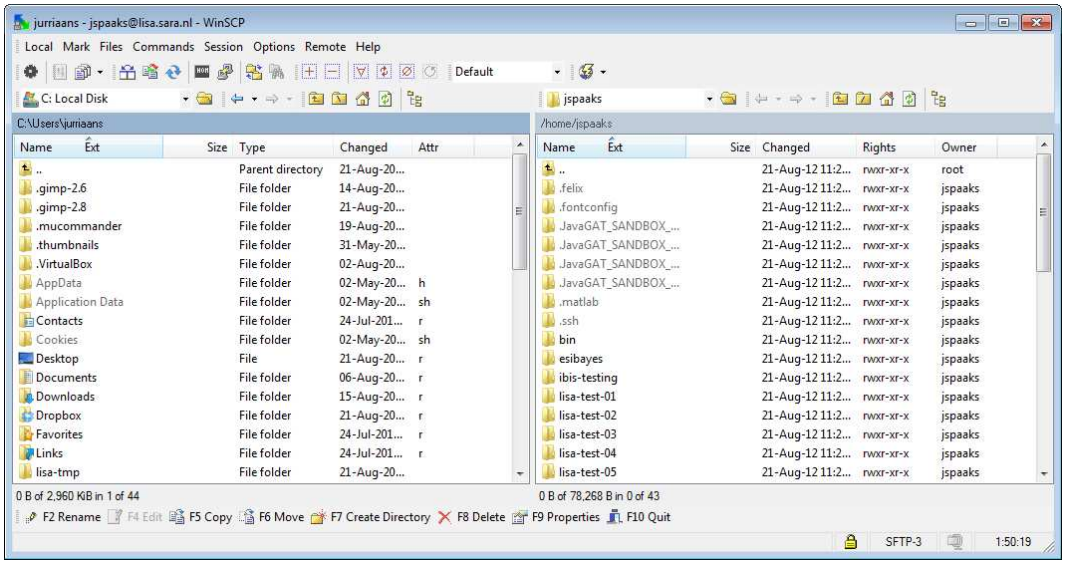


Figure 2.3: WinSCP interface. On the left is my local file system, on the right is my home directory on the remote file system. If this is the first time you connect to LISA, the remote file system should be pretty much empty.

2.4 Issuing commands on the LISA cluster

This section covers how you can issue commands from your local machine, which then get executed by the remote system (LISA). Issuing commands on the LISA cluster can be accomplished through a so-called *terminal emulator* program. A terminal emulator provides you with a prompt at which you can type commands which are then executed on the remote system. It doesn't matter whether the remote system is located just across the street or halfway around the world, you can still operate the remote system through the terminal emulator. However, it is important to note that the LISA cluster, like virtually all clusters[1], runs under the Linux operating system. The commands that you type at the terminal emulator prompt therefore need to be Linux commands, which, as you will see later, are somewhat different from the Windows commands that you may be familiar with.

Let's first download the terminal emulator PuTTY from http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html into your home directory. Double-click the executable to run the program. You should now see the dialog from Fig. 2.4. Under 'Host name (or IP address)', fill in 'lisa.surfsara.nl' and press the 'Open' button. PuTTY first prompts you for your username and then for your password. Fill in your LISA credentials.[2]

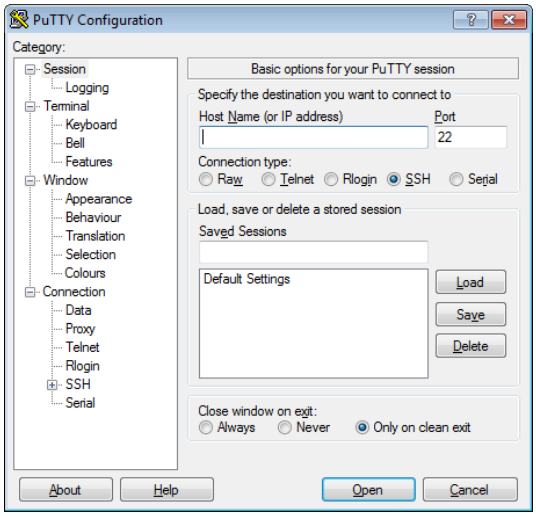


Figure 2.4: PuTTY session dialog box.

You are now remotely logged in to the Linux cluster LISA. Your terminal emulator program should show something like:

```
jspaaks@login2:~$
```

[1] See http://www.top500.org/statistics/ for concurrent statistics on the World's top 500 of supercomputers.

[2] On Linux, bring up a terminal (Ctrl-Alt-t on most distributions) and type in `ssh jspaaks@lisa.surfsara.nl`, substituing your own username in stead of `jspaaks`. Fill in your credentials when prompted.

where `jspaaks` is the username and `login2` is the name of the remote machine. There are, in fact, two machines on which you can log in remotely: besides `login2`, there's also `login1`. For our intents and purposes, it doesn't matter whether you are logged in to `login1` or `login2`, but if you want to change from one to the other, you can do so with:

```
jspaaks@login2:~$ ssh login1
```

When you're done with the terminal, you can close the connection using:

```
jspaaks@login2:~$ logout
```

If you want to find out in which directory you are, you can use the `pwd` command ('pwd' is short for present working directory):

```
jspaaks@login2:~$ pwd
```

which returns:

```
/home/jspaaks
```

It is worth noting that a list of useful commands and terms has been included in the Index section at the back of this document (under 'Linux commands').

So, `pwd` currently returns `/home/jspaaks/`. `/home/jspaaks/` is the user's home directory; it is the Linux equivalent of 'C:\Users\jspaaks' on Windows. Note that on Linux the directory separator is the forward slash '/' sign, whereas Windows uses the backslash '\' sign. You may be wondering why there isn't a 'C:\' in the address returned by `pwd`. The reason is simple: Linux uses '/' instead of 'C:\'. The first forward slash in the address is referred to as the 'file system root'[1]. Under the file system root is a directory 'home', which contains a subdirectory 'jspaaks' in which the user 'jspaaks' keeps all his personal files. As a matter of fact, `/home/jspaaks` is the only place on this file system that user 'jspaaks' is permitted to write anything, meaning that Linux will not allow you to save any files in other users' home directories (although you are allowed to read some of them, depending on how the *permissions* on that directory are set. More about permissions later).

You can view the contents of the current directory using the `ls` (short for 'list') command:

```
jspaaks@login2:~$ ls
```

This should show at least the 'tutorial' folder that we just copied using WinSCP. It is likely that there are a few more files and directories present, but you should leave them as they are.

Changing the current working directory works in a similar way as at the Command Prompt on Windows:

```
jspaaks@login2:~$ cd tutorial
jspaaks@login2:~/tutorial$
```

Note that the prompt includes the location of the current directory: '~' (pronounced: 'tilde') represents the home directory ('/home/jspaaks'). Changing from any location on the file system to the user's home directory goes like so:

```
jspaaks@login2:~/tutorial/a/really/deeply/nested/directory$ cd ~
jspaaks@login2:~$
```

[1] Somewhat confusingly, the account that has Administrator privileges is also referred to as 'root' on Linux systems, but this account has no relation with the root of the file system.

or if you want to move just one directory up:

```
jspaaks@login2:~/tutorial/a/really/deeply/nested/directory$ cd ..
jspaaks@login2:~/tutorial/a/really/deeply/nested$
```

make sure to include the space character in between the `cd` and `..` characters though, otherwise it won't work.

A new directory can be created by the `mkdir` command, for example:

```
jspaaks@login2:~$ mkdir anewdir
```

Note that, contrary to Windows file systems, Linux file systems are case-sensitive. For example:

```
jspaaks@login2:~$ mkdir aNewDir
```

will result in an additional directory:

```
jspaaks@login2:~$ ls
anewdir   aNewDir   tutorial
jspaaks@login2:~$
```

The different approaches that Windows and Linux take regarding case-sensitivity of their respective file systems can lead to errors, especially when copying back and forth between Windows and Linux systems. For example, the contents of `anewdir` and `aNewDir` may get merged when they are copied to a Windows system, since Windows regards the folders as being one and the same. To avoid these kinds of errors, the naming convention on Linux is to use only lower caps names for files and folders, so `anewdir` is preferable to `aNewDir`.

Regardless of your native operating system, your code will be more robust and more portable if you stick to these 2 simple rules when naming your files and directories:

1. don't use spaces;
2. limit yourself to the following subset of characters:
   `abcdefghijklmnopqrstuvwxyz_-.0123456789`,
   and, if you must, `ABCDEFGHIJKLMNOPQRSTUVWXYZ` .[1]

Removing a directory goes like this:

```
jspaaks@login2:~$ rmdir aNewDir
```

or like this if you want to remove multiple directories:

```
jspaaks@login2:~$ rmdir anewdir aNewDir
```

You can delete a file using the `rm` command:

```
jspaaks@login2:~$ rm the-file-you-want-removed.txt
```

or like this if you want to remove multiple files:

```
jspaaks@login2:~$ rm first-file.txt second-file.zip third-file.m etcetera.exe
```

For many commands, you can specify options. Options to a given command must generally be provided directly after the command itself, i.e. before any other argument such as input or output files. The option argument consists of one dash followed by one (case-sensitive)

[1] You'll thank me later!

letter. For example, if you want `ls` to list the contents of the current directory in more detail, you can use the `-l` option (that is the letter *l* for 'long', not the number 1):

```
jspaaks@login2:~$ ls -l
total 4
drwxrwxr-x 2 jspaaks jspaaks 2 Jun 13 13:07 tutorial
```

which gives you, amongst other things, the *permission bits* (i.e. `drwxrwxr-x`), the owner of the item (`jspaaks`), the item's size (`2`) in bytes and the time when the item was last altered (`Jun 13 13:07`). Note that, by default, `ls` sorts the items in a directory based on the item's name, so directories and files will appear intermingled. You can easily tell whether an item is one or the other by looking at the permission bits: if the item is a folder, the first character that appears in the permission bits is the letter `d`. For files, the first character in the permission bits is `-`.

Most Linux commands have at least a few optional arguments. If you want to know more about a particular command's usage, you can use the `man` (short for 'manual') command, which lists all the options for a given program including a short description of what each option does. For example, try:

```
jspaaks@login2:~$ man ls
```

(you can scroll down using the 'Up' and 'Down' arrows. Pressing the 'q' key lets you return to the prompt). If you just want to verify that you remember the command right, you can check by:

```
jspaaks@login2:~$ whatis ls
ls (1)               - list directory contents
```

which will give you a short summary of what `ls` is doing, but without all the technical detail.

In addition to the shorthand notation, some options also have a longer version for improved readibility. The longer version always starts with two dashes instead of one. As an example, the `-h` option makes `ls` list the filesizes in more easily interpretable units such as kB or MB, rather than in bytes:

```
jspaaks@login2:~$ ls -l -h
```

or, combining the shorthand options:

```
jspaaks@login2:~$ ls -lh
```

The longer version of the `-h` option is `--human-readable`, so the complete command becomes:

```
jspaaks@login2:~$ ls -l --human-readable
```

Now that you know a little bit about Linux, let's look at how to manipulate files. `cd` into the 'deeply nested directory' by typing:

```
jspaaks@login2:~$ cd tu
```

if you now press Tab, the *shell* program (i.e. the program that lets you enter commands at the prompt) will automatically complete your command, like so:

```
jspaaks@login2:~$ cd tutorial
```

This autocomplete works because the shell knows that you want to do a `cd`, and since there is only one directory in `~` that starts with `tu`, the shell program knows that you want to `cd` into `tutorial`.

If you `cd` into `a/really/deeply/nested/directory` and list the directory contents, there should be a file called 'with-a-file-in-it.txt'. Because this is an ASCII text file, you can display it in the shell program by using the command `cat`, like so:

```
jspaaks@login2:~/tutorial/a/really/deeply/nested/directory$ cat with-a-file-in-it.txt
```

`cat` is short for 'concatenate', meaning it appends the contents of 'with-a-file-in-it.txt' to whatever is already displayed within the shell. Also note that autocomplete works here as well, but instead of listing any directory names, it will suggest a file from the current directory, since that is the kind of argument that `cat` expects. You will see the following output:

```
jspaaks@login2:~$ cd tutorial/a/really/deeply/nested/directory/
jspaaks@login2:~/tutorial/a/really/deeply/nested/directory$ ls
with-a-file-in-it.txt
jspaaks@login2:~/tutorial/a/really/deeply/nested/directory$ cat with-a-file-in-it.txt
* * * * * * * * * * * * * * * * * * * * *
* *                                     * *
* *  hello world, the classic phrase    * *
* *                                     * *
* * * * * * * * * * * * * * * * * * * * *
jspaaks@login2:~/tutorial/a/really/deeply/nested/directory$
```

(the lines that start with asterisks are actually the contents of 'with-a-file-in-it.txt'.)

Let's now try to move this file to the top directory in the user's home by using the move command `mv` like so:

```
jspaaks@login2:~/tutorial/a/really/deeply/nested/directory$ mv with-a-file-in-it.txt ~
```

The syntax for this command is the command itself, i.e. `mv`, followed by a space, followed by the first input argument, in this case the name of the file that we want to move, i.e. `with-a-file-in-it.txt`, followed by another space, followed by the name of the directory that we want to move it to `~`. Also note that autocomplete works here as well, just type `mv wit` and press Tab to autocomplete.

Let's check that the file really got moved:

```
jspaaks@login2:~/tutorial/a/really/deeply/nested/directory$ cd ~
jspaaks@login2:~$ ls -l
total 6
drwxr-xr-x  4 jspaaks jspaaks    4 Aug 20 11:59 tutorial
-rw-r--r--  1 jspaaks jspaaks  210 Aug 20 15:21 with-a-file-in-it.txt
jspaaks@login2:~$
```

You can also use the `mv` command to rename a file by 'moving' it to a different filename in the same directory like so:

```
jspaaks@login2:~$ ls -l
total 6
drwxr-xr-x  5 jspaaks jspaaks    5 Aug 21 14:45 tutorial
-rw-r--r--  1 jspaaks jspaaks  210 Aug 20 15:21 with-a-file-in-it.txt
jspaaks@login2:~$ mv with-a-file-in-it.txt the-renamed-file.txt
jspaaks@login2:~$ ls -l
total 6
-rw-r--r--  1 jspaaks jspaaks  210 Aug 20 15:21 the-renamed-file.txt
drwxr-xr-x  5 jspaaks jspaaks    5 Aug 21 14:45 tutorial
jspaaks@login2:~$
```

Now suppose we don't want to move a file but we want to copy it. This can be done using the `cp` command. For example:

```
jspaaks@login2:~$ cp with-a-file-in-it.txt copy-of-the-text-file
```

The copy command `cp` expects the file-to-be-copied as its first argument, and the name of the file-to-copy-to as its second argument. Further note that you don't have to specify extensions such as '.txt' in the filename for the operating system to know that `copy-of-the-text-file` is a text file, as you would normally do on Windows. Nevertheless, including the extension in the file name allows easy identification of the type of file, for example when listing the contents of a directory with `ls`, especially when combined with wildcards such as `*`:

```
jspaaks@login2:~/tutorial/a$ ls -l
total 10
-rw-rw-r-- 1 jspaaks jspaaks 16 Aug 22 10:02 file1.txt
-rw-rw-r-- 1 jspaaks jspaaks 16 Aug 22 10:00 file2.txt
-rw-rw-r-- 1 jspaaks jspaaks 16 Aug 22 10:00 file3.txt
-rw-rw-r-- 1 jspaaks jspaaks 13 Aug 22 10:02 file4.m
-rw-rw-r-- 1 jspaaks jspaaks 13 Aug 22 10:01 file5.m
drwxr-xr-x 3 jspaaks jspaaks  3 Aug 20 11:56 really
jspaaks@login2:~/tutorial/a$ ls -l *.txt
-rw-rw-r-- 1 jspaaks jspaaks 16 Aug 22 10:02 file1.txt
-rw-rw-r-- 1 jspaaks jspaaks 16 Aug 22 10:00 file2.txt
-rw-rw-r-- 1 jspaaks jspaaks 16 Aug 22 10:00 file3.txt
jspaaks@login2:~/tutorial/a$
```

`cp` also lets you copy directories, like so:

```
1  jspaaks@login2:~$ cd tutorial
2  jspaaks@login2:~/tutorial$ ls -l
3  total 5
4  drwxr-xr-x 3 jspaaks jspaaks 3 Aug 20 11:56 a
5  drwxr-xr-x 2 jspaaks jspaaks 5 Aug 20 12:07 simple-jobscript
6  jspaaks@login2:~/tutorial$ mkdir another
7  jspaaks@login2:~/tutorial$ cp -R a/* another
8  jspaaks@login2:~/tutorial$ ls -l
9  total 8
10 drwxr-xr-x 3 jspaaks jspaaks 3 Aug 20 11:56 a
11 drwxrwxr-x 3 jspaaks jspaaks 3 Aug 21 14:46 another
12 drwxr-xr-x 2 jspaaks jspaaks 5 Aug 20 12:07 simple-jobscript
13 jspaaks@login2:~/tutorial$ cd another/really/deeply/nested/directory/
14 jspaaks@login2:~/tutorial/another/really/deeply/nested/directory$
```

Line 1 changes the current directory to `tutorial`, line 2 lists its contents, line 6 creates the directory called 'another'. The actual copying is subsequently done in line 7. The complete command consists of the `cp` command, followed by the `-R` option that makes `cp` copy recursively, followed by the source files and folders `a/*` (i.e. everything under `~/tutorial/a`), followed by the destination directory `another`. Line 8 shows the newly created directory, while lines 13–14 show that the copy operation was indeed recursive.

Besides manipulating files and folders, the shell can do a lot more. For example, you can temporarily turn the shell program into an Octave[1] command window, like so:

```
jspaaks@login2:~$ octave
```

The Octave program welcomes you with its default welcome message, followed by a prompt:

```
GNU Octave, version 3.2.4
Copyright (C) 2009 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  For details, type `warranty'.

Octave was configured for "x86_64-pc-linux-gnu".

Additional information about Octave is available at http://www.octave.org.

Please contribute if you find this software useful.
For more information, visit http://www.octave.org/help-wanted.html

Report bugs to <bug@octave.org> (but first, please read
http://www.octave.org/bugs.html to learn how to write a helpful report).

For information about changes from previous versions, type `news'.
```

[1] Octave is a free, open-source clone of MATLAB. See http://www.gnu.org/software/octave/.

```
octave:1>
```

At the prompt, you can type any Octave command (note that the Octave language is almost completely the same as the MATLAB language). For example:

```
octave:1> for k=1:4, disp(['The value of ''k'' = ',num2str(k)]),end
The value of 'k' = 1
The value of 'k' = 2
The value of 'k' = 3
The value of 'k' = 4
octave:2>
```

If you want to return to the normal shell, just type:

```
octave:2> exit
```

From the example above, you can see that typing everything on the command line can be tricky, even for simple tasks. Wouldn't it be great to have an editor of some kind? You've guessed it, the shell can also be turned into a (very) basic editor, like so[1]:

```
jspaaks@login2:~$ nano
```

You can use the nano program to write a simple Octave script, for example the one in Listing 2.1:

Listing 2.1: Example of a simple Octave script in nano.

```
1   GNU nano 2.2.4                New Buffer                                      Modified
2
3  % This is a simple octave script example written in Nano
4
5  % clear any old variables
6  clear
7
8  for k=1:5
9      str = ['The value of ''k'' is ',num2str(k)];
10     disp(str)
11 end
12
13
14
15
16
17 ^G Get Help    ^O WriteOut    ^R Read File   ^Y Prev Page   ^K Cut Text    ^C Cur Pos
18 ^X Exit        ^J Justify     ^W Where Is    ^V Next Page   ^U UnCut Text  ^T To Spell
```

The first line of Listing 2.1 consists of the name and version of the nano program, followed by either the filename (if you opened an existing file) or the string `New Buffer` (if the file has not been saved yet). The string `Modified` is displayed at the right if the user has made any changes. The bottom two lines list a number of keyboard combinations, where the caret symbol `^` represents the Ctrl key, so `^G` means Ctrl-g. After you've typed your script, you can save it by pressing Ctrl-o, typing the filename that you want your script to have, and pressing Enter. You can exit nano by pressing Ctrl-x. If you happen to press Ctrl-x when your script has not been saved yet, nano will ask you if you want to save it before exiting.

Let's check that the script from Listing 2.1 was written to file:

```
jspaaks@login2:~$ cat simple_octave_script.m
```

[1] For more complicated programs, it can be more convenient to do the editing on your local machine using your own tools, and then upload the file to the remote system using WinSCP

```
% This is a simple octave script example written in Nano

% clear any old variables
clear

for k=1:5
    str = ['The value of ''k'' is ',num2str(k)];
    disp(str)
end
jspaaks@login2:~$
```

Now we can call the Octave program using the name of the script as an input argument like so:

```
jspaaks@login2:~$ octave simple_octave_script.m
```

This starts the Octave program, runs the script within it as if you had typed 'simple_octave_script' at the Octave prompt, and returns to the shell:

```
jspaaks@login2:~$ octave simple_octave_script.m
GNU Octave, version 3.2.4
Copyright (C) 2009 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  For details, type `warranty'.

Octave was configured for "x86_64-pc-linux-gnu".

Additional information about Octave is available at http://www.octave.org.

Please contribute if you find this software useful.
For more information, visit http://www.octave.org/help-wanted.html

Report bugs to <bug@octave.org> (but first, please read
http://www.octave.org/bugs.html to learn how to write a helpful report).

For information about changes from previous versions, type `news'.

The value of 'k' is 1
The value of 'k' is 2
The value of 'k' is 3
The value of 'k' is 4
The value of 'k' is 5

jspaaks@login2:~$
```

2.5 Jobscripts and the scheduler

So far, we've executed all our commands on just one machine, 'login2'. As long as the task at hand is a small one, this isn't a problem, but if the task takes a long time to run, it can be advantageous to divide the task into smaller parts, and let each part be computed by a separate machine. The way this works on LISA is as follows: you write a small text file, referred to as a *jobscript*. The jobscript lays out the requirements of the task, such as the number of nodes that are needed, the program that you want to run, and where the necessary data can be found. The jobscript is then sent to a program known as the *scheduler*. The scheduler runs on LISA and manages the requests from all users, such that the computation resources are used in an optimal way, while taking into account things like different priority levels, the number and type of nodes that are needed, and so on.

Listing 2.2 is an example of a simple jobscript.

Listing 2.2: Example of a jobscript.

```
1  #PBS -lwalltime=00:05:00
2  #PBS -lnodes=1:cores8:ppn=1
3  #PBS -S /bin/bash
4
5  echo `date`: job starts
6
7  # let octave execute the disp and pause commands:
8  octave --silent --no-window-system --eval "disp('hello world');pause(90)"
9
10 echo `date`: job ends
11
12 exit
```

This jobscript asks the scheduler for a time slot of 5 minutes (`lwalltime=00:05:00`), and will be using one machine (`lnodes=1`) with 8 cores in its CPU (`cores8`). Only one of these cores will actually be doing something though, because the number of processes per node is just 1 (`ppn=1`). The next line tells the scheduler that the rest of the jobscript is written in a scripting language called 'bash', which is a very common scripting language on Linux. Incidentally, you already know it, since the shell program you have been using is in fact bash. The script then echoes the current date and time plus the message ': job starts' to the standard output (more about standard output later). Line 8 is the core of the script, in that it specifies what program needs to be run. In our case, it says that it wants to start an instance of the Octave software (`octave`) and that this instance needs to be started with the `--silent` and `--no-window-system` options, such that it will not display the usual welcome message and that it will not use any of the graphical output methods (i.e. Octave's `figure` command will not yield the usual output). The string following the `--eval` option specifies the Octave command that will be run by the Octave software. Here, it is simply the `disp('hello world')` command, but it could be any string that qualifies as valid Octave code, including function names and script names. (In a minute, it will become apparent why I added the `pause(90)`). The `disp` command always writes to the standard output, which normally equates to saying that it outputs to your screen, but on the LISA system it actually is a file which we will check later. The script then echoes the current date and time plus the message ': job ends' to the standard output, before exiting the script.

OK, so now that we have a jobscript file (which I saved as '~/tutorial/simple-batch/jobscript.pbs'—LISA uses the OpenPBS/Torque scheduler[1], so I usually save my jobscripts as *.pbs), let's send it to the scheduler with the `qsub` command (`qsub` is an abbreviation for 'submit to the queue', the queue in question being the scheduler's):

```
jspaaks@login2:~/tutorial/simple-batch$ qsub jobscript.pbs
6388734.batch1.irc.surfsara.nl
jspaaks@login2:~/tutorial/simple-batch$
```

As you can see above, `qsub` prints a number `6388734` to the shell. This number is referred to as the *job id*. You can check the status of the job by typing the `qstat` command, followed by the job id, like so:

```
jspaaks@login2:~/tutorial/simple-batch$ qstat 6388734
Job id                    Name             User             Time Use S Queue
------------------------- ---------------- ---------------- -------- - -----
6388734.batch1            jobscript.pbs    jspaaks                 0 Q express
```

[1] http://www.adaptivecomputing.com/products/open-source/torque/

```
jspaaks@login2:~/tutorial/simple-batch$
```

or, you can ask for the status of all jobs that are owned by a specific user, using `qstat`'s `-u` option, like so:

```
jspaaks@login2:~/tutorial/simple-batch$ qstat -u jspaaks
Job ID            Username Queue    Jobname         SessID NDS TSK Memory Time  S Time
----------------- -------- -------- --------------- ------ --- --- ------ ----- - -----
6388734.batch1.i jspaaks  express jobscript.pbs     --    1   1    --  00:05 Q   --
```

This is where the `pause(90)` comes in: it is to make sure that the job runs sufficiently long for you to check the job's statistics, because the LISA system can't show any statistics for jobs that are finished already. Once you've seen how the `qstat` command works, you can remove the `pause(90)` if you like.

The `S` column lists the status of the job. In this case, the job is queued, hence its status is listed in the table as `Q`. Besides queued, it can be `R`unning, `C`ompleted, or `H`eld. (There are a few more statuses which are more obscure, see `man qstat`). `qstat`'s `-n` option shows the nodes that are allocated for your job (here: 'gb-r2n14'):

```
jspaaks@login2:~/tutorial/simple-batch$ qstat -u jspaaks -n

batch1.irc.surfsara.nl:
                                                                  Req'd  Req'd   Elap
Job ID            Username Queue    Jobname         SessID NDS TSK Memory Time  S Time
----------------- -------- -------- ---------------- ------ --- --- ------ ----- - -----
6388734.batch1.i  jspaaks  express jobscript.pbs 15291    1   1    --  00:05 R 00:00
   gb-r2n14/0
jspaaks@login2:~/tutorial/simple-batch$
```

You can look up statistics for a given node at https://ganglia.surfsara.nl/?r=hour&cs=&ce=&s=by+name&c=LISA%2520Cluster&tab=m&vn=, for example if you want to check its performance (see also Fig. 2.5).

Because `6388734` is such a tiny job in terms of walltime and number of nodes, the scheduler adds the job to the 'express queue'. Depending on the requirements layed out in the jobscript, your job may end up in one of 3 different queues:

1. the express queue
2. the serial queue
3. the parallel queue

Your job ends up in the express queue if it asks for less than 5 minutes walltime and doesn't use more than one node. If the system is not too busy, jobs in the express queue are usually executed without delay. The express queue is primarily meant for development work. If your job asks for more than 5 minutes on one node, you end up in the serial queue. Despite its name, you can still run parallel jobs, but they will be limited to the cores of the one node that was allocated to you (in LISA's current configuration, that means a maximum of 16 parallel processes). Your job ends up in the parallel queue if you ask for more than one node. Jobs that are in the serial or parallel queues can take a long time to start, but when they finally do, you can unleash the full computing power of the cluster, without performance loss due to other users.

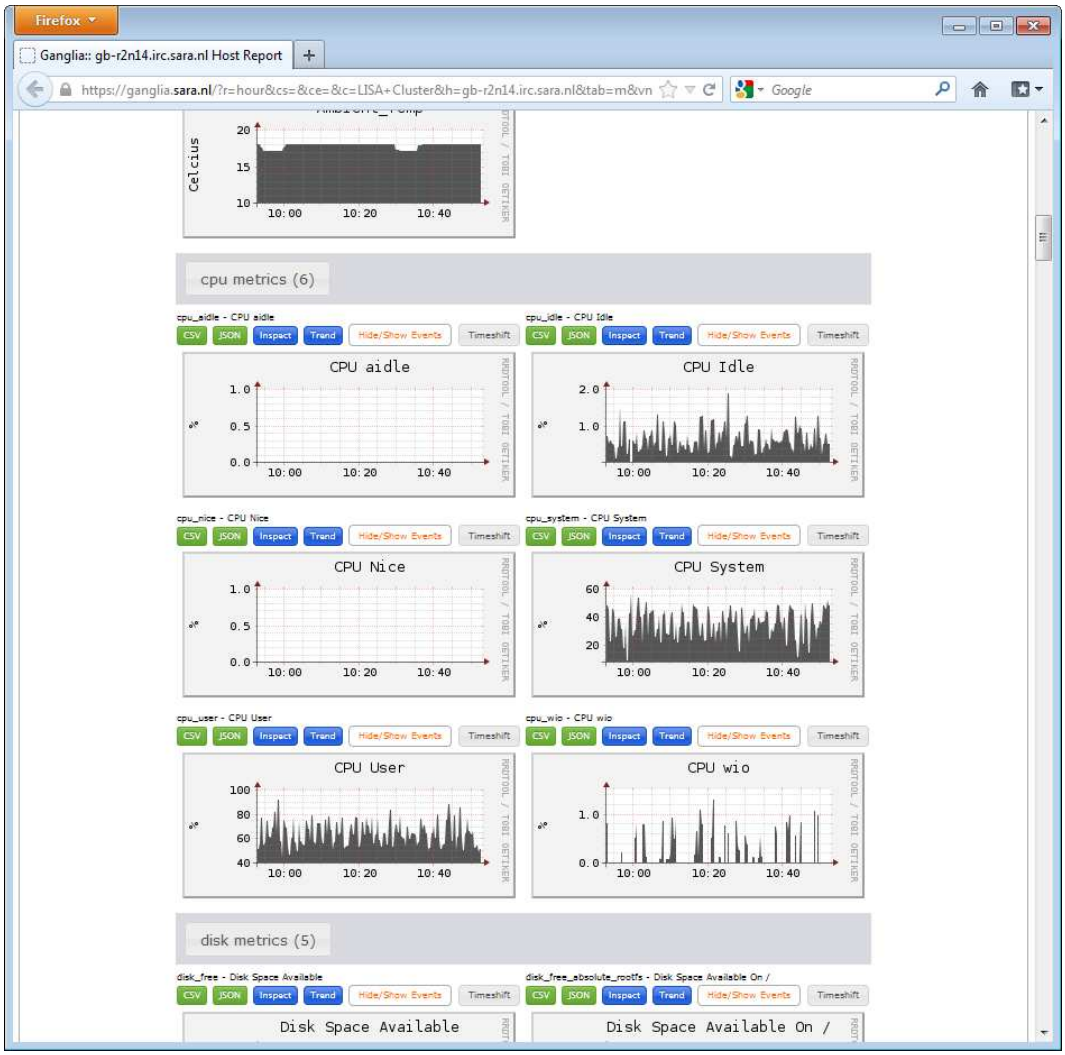


Figure 2.5: Ganglia visualizes a wide array of performance indices for the LISA system. The plots can be viewed in a web browser such as Firefox.

If you need to develop software that requires testing on multiple machines, you can start an interactive session using the `-I` option as follows:

```
jspaaks@login2:~/tutorial$ qsub -I -lnodes=2:cores8 -lwalltime=00:59:00
qsub: waiting for job 6388157.batch1.irc.surfsara.nl to start
```

The above command asks for a 59-minute session with 2 nodes of the core8 type. Note that the information that is normally passed to `qsub` through the first few lines in a jobscript (i.e. the lines that start with `#PBS`) must now be entered as options. After a delay, you will get a prompt on a different machine, in this case 'gb-r7n32':

```
qsub: job 6388157.batch1.irc.surfsara.nl ready

jspaaks@gb-r7n32:~$
```

At this prompt, you can manually enter your commands, just like you were doing earlier, except you're no longer on 'login2' anymore, but on one of the computing nodes within the cluster. Either one of the login nodes may have many users on them at any given time, whereas the nodes that you get through the `qsub` command are exclusively yours (at least for a limited time). This means that you can take full advantage of the computing power and available memory.

When your time is up, the following message will appear:

```
jspaaks@gb-r7n32:~$ echo $PBS_O=>> PBS: job killed: walltime 3575 exceeded limit 3540
```

and you will return to 'login2' (or 'login1' if that's where you were before).

The `showq` command lets you view (your part of) the queue, for example:

```
jspaaks@login2:~/tutorial/simple-batch$ showq -u jspaaks
ACTIVE JOBS--------------------
JOBNAME            USERNAME      STATE  PROC   REMAINING            STARTTIME

6388079             jspaaks    Running     8    00:01:00  Thu Aug 23 15:41:24

     1 Active Job     6444 of 6684 Processors Active (96.41%)
                       617 of  647 Nodes Active      (95.36%)

IDLE JOBS----------------------
JOBNAME            USERNAME      STATE  PROC     WCLIMIT            QUEUETIME


0 Idle Jobs

BLOCKED JOBS----------------
JOBNAME            USERNAME      STATE  PROC     WCLIMIT            QUEUETIME


Total Jobs: 1   Active Jobs: 1   Idle Jobs: 0   Blocked Jobs: 0
jspaaks@login2:~/tutorial/simple-batch$
```

Also, `showq` shows whether the cluster is very busy or not (Friday afternoons are usually the busiest; Sundays, Mondays and Tuesdays are quiet in comparison). The statistics displayed by `showq` are updated every 30 seconds or so.

Some other commands that come in handy from time to time are:

```
jspaaks@login2:~/tutorial/simple-batch$ qdel 6388079
```

This deletes job `6388079` from the queue. Very useful if you accidentally requested too many nodes, or when you realize you made a mistake in your program but the job was submitted already. It's not possible to delete a job that was not submitted by you, so you don't have to worry about accidentally deleting other people's jobs if you type the job id wrong.

`checkjob` gives a more detailed overview of the job:

```
jspaaks@login2:~/tutorial/simple-batch$ checkjob 6388672


checking job 6388672

State: Running
Creds:  user:jspaaks  group:lisa_uva  class:express  qos:DEFAULT
WallTime: 00:00:00 of 00:01:00
SubmitTime: Thu Aug 23 17:17:24
  (Time Queued  Total: 00:00:01  Eligible: 00:00:01)

StartTime: Thu Aug 23 17:17:25
Total Tasks: 1

Req[0]  TaskCount: 1  Partition: DEFAULT
Network: [NONE]  Memory >= 0  Disk >= 0  Swap >= 0
Opsys: [NONE]  Arch: x86_64  Features: [q_express][cores8]
Allocated Nodes:
[gb-r7n32:1]


IWD: [NONE]  Executable:  [NONE]
Bypass: 0  StartCount: 1
PartitionMask: [ALL]
Flags:       BACKFILL RESTARTABLE

Reservation '6388672' (00:00:00 -> 00:01:00  Duration: 00:01:00)
PE:  1.00  StartPriority:  -8641

jspaaks@login2:~/tutorial/simple-batch$
```

Finding out when your job is scheduled to start can be accomplished with the `showstart` command like this (although the estimate is quite inaccurate in my experience):

```
showstart 6388079
```

If you want to check how much computing time your **acc**ount has **use**d, you can use the `accuse` command, or the **acc**ount **info**rmation (`accinfo`) command.

This concludes the Linux tutorial part of this manual. The next chapters cover the general concept of parallel optimization (Chapter 3) and the MMSODA software (Chapter 4).

Chapter 3

Parallel optimization

Optimization algorithms such as SCEM-UA (Vrugt et al., 2003b), SODA (Vrugt et al., 2005a), Differential Evolution (Storn and Price, 1997), or DREAM (Vrugt et al., 2009) repeatedly sample the parameter space. In this chapter, we look at how such a sampling algorithm may be parallelized, and we introduce a law that describes the speedup that can be expected from parallelizing a particular piece of software.

Most parameter optimization algorithms sample the parameter space in generations of independent samples. When the optimization is implemented as a *serial* or *sequential* algorithm, the members of a generation are evaluated one after the other. For example, if the tasks that need to be evaluated each consist of one parameter set and there are 10 such tasks, the walltime that is needed to evaluate these tasks is equivalent to the sum of the CPU time that individual tasks need (Figure 3.1). In contrast, if the optimization algorithm employs a parallel strategy for evaluating the tasks, and the tasks are started simultaneously on 10 CPUs, then the walltime is determined by the CPU time of the slowest task (including the overhead introduced by the parallelization). The maximum speed-up S that be gained from executing a task in parallel is expressed by *Amdahl's Law*:

$$S = \frac{1}{\alpha} \tag{3.1}$$

in which α is the fraction of the program that cannot be parallelized.

For the example in Figure 3.1, each task is about 10% of the total walltime when executed sequentially, so α is about 0.1 (not counting the overhead due to parallelization), therefore the theoretical maximum speedup is about 10x. When executed in parallel, a relatively large part of the time is spent in communication ('parallelization overhead'). When this is the case, the application is said to be *fine-grained*. In contrast, if a parallel implementation of a program spends most of its time on calculation (as opposed to on communication), it is said to be *coarse-grained*. Most Monte-Carlo based optimization problems are coarse-grained, since the model CPU time is usually at least a few seconds, or sometimes even hours—in any case, much less than the time needed to send a parameter set over the network. Problems for which the communication time is negligible compared to the calculation time are said to be *embarassingly parallel*.

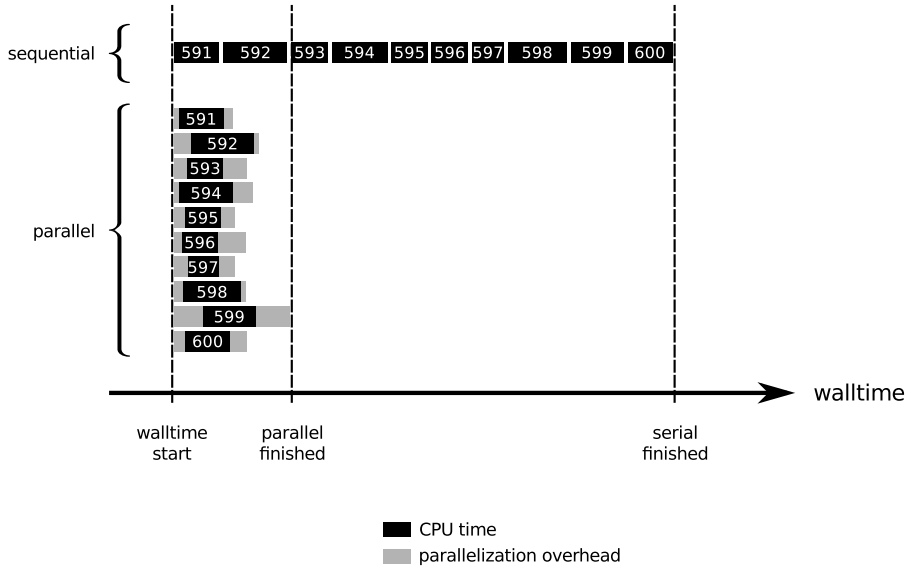


Figure 3.1: Walltime needed to complete 10 tasks 591–600 when evaluated sequentially or in parallel. The actual speedup that is achieved in parallelizing this problem is less than the theoretical maximum speedup due to parallelization overhead.

3.1 The Master-Worker paradigm

For Monte-Carlo type optimization, the Master-Worker paradigm (see Fig. 3.2; sometimes also referred to as Master-Slave paradigm) is the most common. In the Master-Worker paradigm, one of the available nodes is assigned the role of Master, while all the other nodes assume Worker roles.

3.1.1 Master side

The Master node is responsible for keeping track of the status of all Worker nodes with regard to what program a given Worker should run next, as well as to the logistics of the optimization, i.e. whether all nodes are connected, which nodes are busy calculating, which have finished, which are waiting for a new task, etc. Furthermore, the Master node is the brains of the optimization algorithm: the Master decides which tasks should be evaluated next. At the very beginning of the optimization, the Master typically checks whether all the nodes are online. After that, the Master will send each node all data that the Worker needs to complete its tasks. This typically includes things like the observations against which the model result will be compared, as well as the model's initial and boundary conditions—basically everything that is constant for all tasks. The Master also sends the model structure itself. Once all Workers have received the data and the model structure, the Master node determines what parameter combinations need to be sampled first. This is

entirely dependent on which algorithm is used for the optimization. In any case, the Master compiles a list of all parameter combinations that need to be evaluated (i.e. for which the model structure needs to be run). It then distributes these parameter combinations over the available Workers, and waits for the Workers to start returning results. A result is usually in the form of an objective score or likelihood function value, but can in principle be any variable, or even a collection of variables.

3.1.2 Worker side

On the Worker side, things are pretty simple: when the Worker starts, it loads all the data that it received from the Master, after which it goes into a never-ending loop. The never-ending loop consists of just a few components: first, it waits for a new task, i.e., a new parameter combination. After it receives a parameter combination, it runs the model structure with it. After the model finishes, the Worker will typically run some sort of objective function to calculate the likelihood function value, for instance by comparing it to observations (which the Master sent to the Worker at the very beginning). The objective function's result is then sent back to the Master node, and the Worker returns to the beginning of the never-ending loop where it either continues with the next task, or waits until it receives a new one.

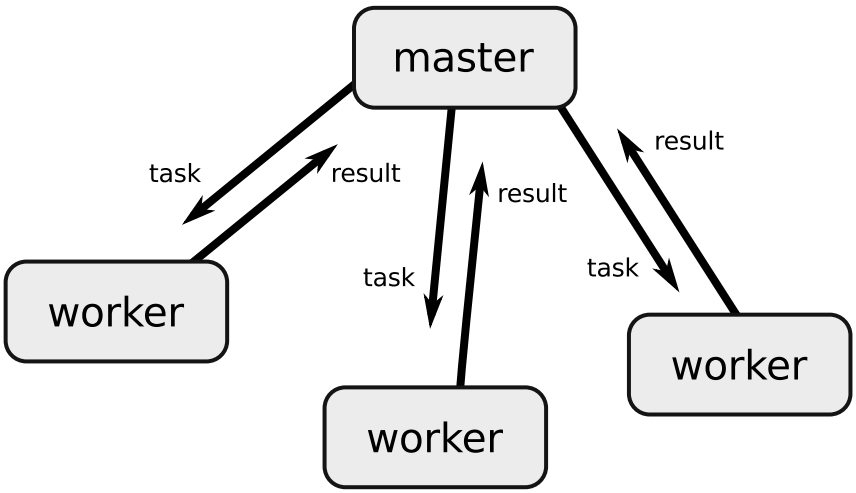


Figure 3.2: Message passing between a master node and its workers.

Chapter 4

The MMSODA Toolbox for MATLAB

There are various softwares that implement the Master-Worker paradigm. The most commonly used Master-Worker software is probably MPI, which stands for *Message Passing Interface*. Most cluster computers have some form of MPI installed. We will use the GNU version of OpenMPI, since this is the default MPI package at the LISA cluster computer.

We have developed a parallel MATLAB version of SODA (e.g. Vrugt et al., 2005a) that makes use of MPI. The software package is called 'The MMSODA Toolbox for MATLAB', or MMSODA for short (MMSODA stands for MATLAB-MPI-SODA). In this chapter, we will look at how to set it up.

MMSODA offers the functionality of a number of previously separate softwares, namely SCEM-UA (Vrugt et al., 2003b), SODA (Vrugt et al., 2005a,b; Clark and Vrugt, 2006), MOSCEM-UA (Vrugt et al., 2003a), multi-objective SODA (Vrugt et al., 2008), the MPITB-parallel version of SCEM-UA implemented in Octave (Vrugt et al., 2006b), and the MPITB-parallel version of SODA implemented in Octave (Vrugt et al., 2006a). Additionally, MMSODA offers a parallel version of multi-objective SCEM-UA, and a parallel version of multi-objective SODA, both of which did not exist previously. Moreover, MMSODA does not use Octave when running in parallel, because Octave does not evaluate code as quickly as does MATLAB. MMSODA circumvents (in a legal fashion) the license requirements that are often an impediment to parallel computation by compiling the MATLAB code into a binary which can be run without any license. Compiling the binary, however, does require a license, both for the MATLAB program itself, as well as for the MATLAB Compiler Runtime Toolbox. Fortunately, the required licenses are available on most cluster environments targeting a scientific and engineering audience.

In short, the acronyms mentioned above mean that: MMSODA can do parameter tuning with or without intermediate state updating by an ensemble Kalman Filter; that MMSODA supports both single-objective and multi-objective optimization; and that the optimization can be run either sequentially on a local machine, or in parallel on a cluster computer.

The serial/parallel capability is particularly attractive, since it allows the users to set up

their optimizations locally on their own machines, thus ensuring a familiar development environment without the need to make the code compatible with Octave syntax. When the user finishes setting up the optimization, running it on a cluster computer is simply a matter of copying the relevant directory to the cluster storage using standard tools (e.g. WinSCP) and compiling the software by executing a script that comes with the software. Furthermore, MMSODA is fully documented with HTML documentation which can be accessed in the same way as MATLAB's built-in commands, namely through the `doc` command.

The remainder of this chapter explains how to set up increasingly sophisticated optimizations within the MMSODA framework. Let's start off with a single-objective SCEM-UA optimization of a benchmark function `calcLikelihood()`, and let's not do anything in parallel just yet.

4.1 MMSODA in 'bypass' mode; sequential execution

▶ 4. You should have received 'esibayes.zip' with this document. Unzip it into a directory on your local machine.

As you can see, the 'esibayes' directory contains a number of subdirectories. Among these, the 'mmsoda-toolbox' directory is probably the most important: it contains the MATLAB toolbox that this chapter is about. Most of the other directories have names like 'soda-so-something' or 'bypass-mo-something'. The specific meaning of these names will become clear later; for now, it suffices to say that they are example projects of how to use the MMSODA toolbox for MATLAB with different models and different configurations. Finally there are two more directories, one labeled 'template-project', and one labeled 'other'. The former contains a rudimentary structure of an MMSODA project, whereas the latter contains some files that we will be using later on.

▶ 5. Let's get started by making a new directory next to the `mmsoda-toolbox` directory. Let's call this directory `example1`. Change directory into `example1`, and create three directories in it called `data`, `model`, and `results`. (We won't always use all three directories, but MMSODA expects all three to be present regardless of whether they are used). So now you have a directory somewhere on your local storage device that has at least the following subdirectories:

- `esibayes/example1`
- `esibayes/example1/data`
- `esibayes/example1/model`
- `esibayes/example1/results`
- `esibayes/mmsoda-toolbox` (which includes a bunch of subdirectories and subsubdirectories that are not relevant for the moment).

▶ 6. Open MATLAB and set your working directory to `example1`.

▶ 7. Before we can use the functionality provided by the MMSODA Toolbox for MATLAB, we need to tell MATLAB about its existence by adding the main directory to the MATLAB search path. You must do this using the `addpath` command as follows:

`>> addpath('s')`, in which `s` indicates the location of the MMSODA directory. For example, it could be:
`>> addpath('C:\Users\jspaaks\esibayes\mmsoda-toolbox')`
on a Windows machine or
`>> addpath('/home/jspaaks/esibayes/mmsoda-toolbox')`
on Linux.

► 8. At the MATLAB prompt, type:
`>> mmsoda --docinstall`
to complete the MMSODA setup. In principle, you only have to run this command once per MATLAB session, as long as you do not change the location of the 'mmsoda-toolbox' directory on your storage.

► 9. Test whether everything works as it should by typing:
`>> doc mmsoda`
at the MATLAB command prompt. This should bring up MATLAB's help browser. Click on the link 'View HTML documentation for this function in the help browser'. You should now see an overview of the functions comprising the MMSODA Toolbox for MATLAB.

► 10. Spend at least 12 minutes to browse through the documentation. In any case, make sure to read the documentation on 'mmsoda.m'.

The function that we want to maximize implements the double-normal probability distribution:

$$p = \frac{1}{2} \cdot \frac{1}{\sqrt{2\pi\sigma_1^2}}\, e^{-\frac{1}{2}\left(\frac{x-\mu_1}{\sigma_1}\right)^2} + \ldots \\ \frac{1}{2} \cdot \frac{1}{\sqrt{2\pi\sigma_2^2}}\, e^{-\frac{1}{2}\left(\frac{x-\mu_2}{\sigma_2}\right)^2} \tag{4.1}$$

with $\mu_1 = -10$, $\sigma_1 = 3$, $\mu_2 = 5$, $\sigma_2 = 1$, respectively. The parameter that is optimized (or, equivalently, whose probability distribution we will estimate by means of the MMSODA Toolbox for MATLAB) is x. For example, for $x = 4.5$, $p = 0.1760$.

Because MMSODA expects the objective function to return a log-likelihood l, we must actually take the natural logarithm of p as the objective score:

$$l = \ln(p) \tag{4.2}$$

4.1.1 Creating the 'constants.mat' and 'conf.mat' files

Before we actually start writing any code for this objective function however, let's first create the 'conf.mat' and 'constants.mat' files that are always needed for running `mmsoda()`.

► 11. Start a new text file in the MATLAB editor and save it as 'makeconf.m' in the current working directory.

► 12. At the first line in 'makeconf.m', add the following:
`function makeconf()`

Now we need to edit the contents of 'makeconf.m' as follows.

► 13. At the MATLAB prompt, type
`doc mmsoda`
and bring up the HTML documentation for the `mmsoda()` function.

Near the bottom of the documentation, there is an overview of the configuration variables that must be specified for a given type of optimization. For our double-normal example, we will use MMSODA in 'bypass' mode. This mode is used when the log-likelihood can be estimated directly from the parameter vector, without the need to run a (dynamic) model structure.

► 14. If you look in the table with the configuration variables, you'll see that only 5 variables are required for running MMSODA in 'bypass' mode. These are `modeStr`, `objCallStr`, `parNames`, `parSpaceHiBound`, and `parSpaceLoBound`. Make sure you understand the description for each of these.

► 15. Return to 'makeconf.m' and add the following:
```
modeStr = 'bypass';
objCallStr = 'calcLikelihood';
parNames = {'x'};
parSpaceHiBound = [10];
parSpaceLoBound = [-30];
```

With the above settings we specify that we want MMSODA to do a bypass run, in which the function 'calcLikelihood.m' (which we will create shortly) is optimized. `calcLikelihood` has one tunable parameter, `x`. The bounds that we set on the search for the optimal value of `x` are `[-30,10]`.

► 16. At the last line in 'makeconf.m', add the following:
```
save('./results/conf.mat')
```

► 17. Save and close 'makeconf.m'.

Next, we need to create 'constants.mat' by a similar procedure.

► 18. Create a new m-file in the current working directory called 'makeconstants.m'.

► 19. At the first line in 'makeconstants.m', type:
```
function makeconstants()
```

Now we need to assign the constants, i.e. the variables that `calcLikelihood` needs in order to calculate the log-likelihood according to equations 4.1–4.2.

► 20. In 'makeconstants.m', add:
```
parMu1 = -10;
parSigma1 = 3;
parMu2 = 5;
parSigma2 = 1;
```
i.e. the two means and two standard deviations for the double normal distribution.

► 21. At the last line in 'makeconstants.m', add
`save('./data/constants.mat')`

► 22. Save and close 'makeconstants.m'.

Finally, we need to create the objective function m-file that implements equations 4.1 and 4.2.

4.1.2 Creating the objective function m-file

► 23. Create a new m-file, called 'calcLikelihood.m' and save it in the subdirectory './model'.

► 24. Open './model/calcLikelihood.m'. MMSODA uses a standardized way of passing the input and output arguments to and from the objective function, so the first line is always exactly the same (with the exception of the name of the function `calcLikelihood`, which may vary), like so:
`function objScore = calcLikelihood(conf,constants,modelOutput,parVec)`

► 25. As a second line, type:
`mmsodaUnpack()`
This function uses the information from the input arguments to construct the variable `x` and to assign it a value based on the value of `parVec`. Similarly, it uses information from the `constants` variable to construct the model constants and to assign them the correct values.

Now that we have `parMu1`, `parSigma1`, `parMu2`, `parSigma2`, and `x` we can calculate the probability density `dens` as follows:

```
dens = (1/(sqrt(2*pi*parSigma1^2))*exp(-(1/2)*((x-parMu1)/parSigma1)^2) + ...
        1/(sqrt(2*pi*parSigma2^2))*exp(-(1/2)*((x-parMu2)/parSigma2)^2))/2;
```

► 26. Add this calculation to your `calcLikelihood` function.

► 27. Don't forget that MMSODA expects a log-likelihood however, so as a final line in `calcLikelihood`, add:
`objScore = log(dens);`

► 28. Save and close 'calcLikelihood.m'.

4.1.3 Running the optimization locally

► 29. Make sure that the current working directory is the 'example1' directory. At the MATLAB command prompt, type:
`>> makeconf()`
and check that a new file 'conf.mat' is created in subdirectory './results'.

► 30. At the MATLAB command prompt, type:
`>> makeconstants()`
and check that a new file 'constants.mat' is created in subdirectory './data'.

► 31. At the command prompt, type `clear` to clear the workspace if there are any variables in it.

► 32. Now, we are ready to run the optimization. At the MATLAB command prompt, type:
`>> [evalResults,critGelRub,sequences,metropolisRejects,conf] = mmsoda();`
and wait for the optimization to finish. Input arguments to `mmsoda` are not required, since `mmsoda` knows to look in './results/conf.mat' for the configuration, in './data/constants.mat' for the model constants, and in './model' for the model functions and objective functions.

4.1.4 Interpreting the results

Once the optimization finishes, you should have 5 variables in your workspace: `evalResults`, `critGelRub`, `sequences`, `metropolisRejects`, and `conf`. Refer to the `mmsoda()` documentation for a description of what these variables are and how they are laid out.

Let's explore the results by making a histogram of the occurence of certain parameter values. You could simply use MATLAB's built-in `hist()` function to do this, but it is often more convient to use a specialized function that comes pre-packaged with MMSODA like so:

```
>> mmsodaSubplotScreen(1,2,1);
>> mmsodaMargHist(conf,evalResults);
```

► 33. Refer to the documentation of these two functions to see how they work and use the optional `'nHistory'` argument to show the marginal histogram based on the last quarter of the `evalResults` record. Your result should look like Fig. 4.1.

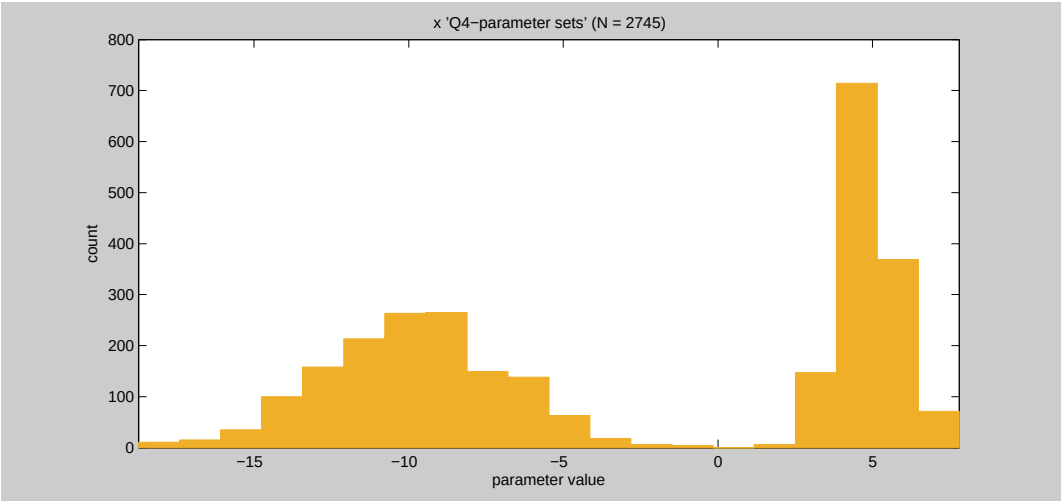


Figure 4.1: Histogram for the Q4 parameter sets for the double-normal model.

► 34. Refer to the documentation on how to use `mmsodaPlotSeq`. Create another figure on the right side of your screen, in which you visualize the record of sequences. Use the options to hide the samples that were rejected as part of the Metropolis scheme. You result should look like Fig. 4.2.

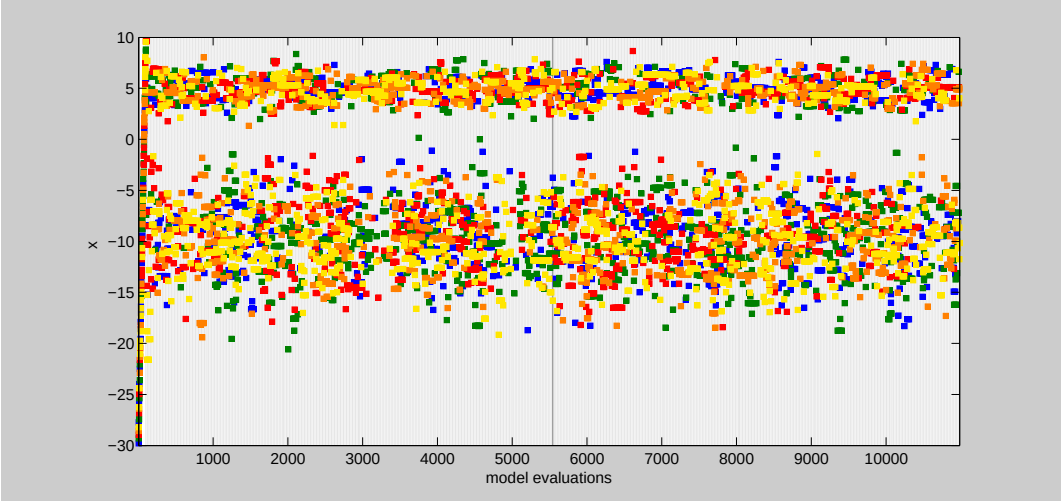


Figure 4.2: History of samples taken from the 1-D parameter space by SCEM-UA during optimization of the double-normal model.

► 35. When you are satisfied with the way you set up MMSODA locally, you can make preparations for running it in parallel on the LISA cluster computer. Running on LISA requires a so-called 'Makefile' as well as a jobscript, both of which are tricky to write yourself. Therefore, the MMSODA Toolbox for MATLAB comes with a function that helps you generate the correct files by asking a series of questions. At the MATLAB prompt, type:
`>> mmsodaPrepParallelFiles()`
and use the following information to answer its questions:

1. the optimization will run on one of the login nodes;
2. we want not much verbal feedback from the program;
3. we don't want to use all the cores on the login node;
4. we want to start 4 processes;
5. we don't want to save the timing information.

(Note that `mmsodaPrepParallelFiles()` indicates the default answer with brackets, and that you can accept the default by simply pressing Enter.)

When `mmsodaPrepParallelFiles()` finishes, it prints a message in the command window that tells you what file it has just created. This file should be located in the current working directory. We will use it shortly to start the optimization on the cluster. You can take a look a its contents in Notepad or a similar program, but make sure not to change anything. Besides the jobscript, there should also be a new file called 'Makefile'. Just like the jobscript, this is also a plain text file, so you can view its contents in Notepad as well. Again, make sure not to accidentally change anything.

4.2 MMSODA in 'bypass' mode; parallel execution

► 36. Use WinSCP or the alternative program of your choice to copy the 'example1' and 'mmsoda' directories, including all of their contents, to your storage on the cluster, i.e. anywhere under the '/home/<username>/' directory.

When you are satisfied with the way you set up MMSODA locally, you can run it on the LISA cluster computer. In order to do so, we must first compile the software into a so-called 'binary' or 'executable'. You do not need to worry about how this works in detail, it is just a matter of running the Linux `make` command. `make` looks for a file called 'Makefile' that was just created by `mmsodaPrepParallelFiles()`. Based on the contents of 'Makefile', `make` collects all the relevant software (your model files, your objective functions, the MMSODA code, as well as the code that enables communication between the Master and the Workers) and creates two files that are necessary to run your code within MMSODA using multiple cores.

4.2.1 Compiling MMSODA and your model code into a binary

► 37. Use PuTTY to start an SSH connection to the LISA cluster.

► 38. In the PuTTY terminal, load the MATLAB program and MPI programs by typing:
`module load matlab`
This command will not give any feedback on the success or otherwise of the command, but you could check by typing the following command:
`module list`
which should now include MATLAB. Next, type
`module load openmpi/gnu`
to load the MPI software.

► 39. Use the `cd` command to set 'example1' as your current directory if you hadn't already done so.

► 40. Now we are ready to compile. At the terminal, type:
`make`
You should see some text scrolling over your screen—it takes about 60 seconds or so to complete. The `make` command looks for a file called 'Makefile' in the current directory, and uses the information in it to correctly build the binary 'matlabprog' and the library that it needs, called 'libmmpi.so'.

► 41. After `make` finishes, list the directory contents with `ls -l` and verify that you now have two extra files 'matlabprog' and 'libmmpi.so'.

Starting the optimization requires that we adjust the 'permission bits' for the 'runmmsoda.sh' file that was just created by `mmsodaPrepParallelFiles()`. Permission bits indicate what a specific user is allowed to do with a particular file. (You may know the same concept from Windows, where you can sometimes have 'Read-only' versions of a file). The permission bits are listed as the first 10 columns in the output from `ls -l`:

```
jspaaks@login1:~/esibayes/example1$ ls -l
total 204
drwxr-xr-x 2 jspaaks jspaaks     26 Jan 16 16:09 data
-rwxrwxr-x 1 jspaaks jspaaks 172792 Jan 18 11:57 libmmpi.so
-rw-r--r-- 1 jspaaks jspaaks    176 Jan 16 16:02 makeconf.m
-rw-r--r-- 1 jspaaks jspaaks    120 Jan 16 15:06 makeconstants.m
-rw-rw-r-- 1 jspaaks jspaaks   1357 Jan 16 16:10 Makefile
-rwxrwxr-x 1 jspaaks jspaaks  11969 Jan 18 11:57 matlabprog
drwxr-xr-x 2 jspaaks jspaaks     29 Jan 16 16:09 model
drwxr-xr-x 2 jspaaks jspaaks   4096 Jan 16 17:36 results
-rw-r--r-- 1 jspaaks jspaaks    580 Jan 18 13:26 run-mmsoda.sh
jspaaks@login1:~/esibayes/example1$
```

For 'run-mmsoda.sh', the permissions are set to `-rw-r--r--`. The first character `-` indicates that 'run-mmsoda.sh' is a file (as opposed to a `d` which would indicate a directory). Characters 2, 3 and 4 (`rw-`) indicate what you, the currently logged-in user, is allowed to do with 'run-mmsoda.sh'. Currently, you are allowed to read from (`r`) and write to (`w`) 'run-mmsoda.sh'. The `-` character from the fourth column of `ls -l` indicates that you are currently not allowed to execute 'run-mmsoda.sh' as a script.

► 42. Change the permission bit for 'run-mmsoda.sh' by typing at the prompt:
`chmod u+x run-mmsoda.sh`
In normal English, this command translates to "`Ch`ange the `mod`e of file 'run-mmsoda.sh' by adding (`+`) the executable permission (`x`) for the current user (`u`)". Check that the permissions have changed to `-rwxr--r--`.

► 43. Now we are ready to start the optimization in parallel on the login node. At the beginning of the optimization, you will see a lot of text scrolling over your terminal screen which at this stage probably does not make much sense. Don't worry if you don't understand it—we will look at it in greater detail later. Eventually, you'll get messages along the lines of 'Evaluating parameter sets 1-100', 'Evaluating parameter sets 101-120', etc. Start the optimization by typing at the terminal:
`./run-mmsoda.sh`
(Don't omit the `./` at the beginning, otherwise it won't work.)

► 44. After the optimization finishes, the results need to be transferred to your local system. Copy the contents of 'example1/results/' from the remote system to 'example1/results/' on your local machine (you can overwrite the old files in 'example1/results/' on your local machine if you want).

► 45. On your local machine, clear the workspace, and load MMSODA's output variables using:
`>> load('./results/bypass-so-results.mat')`
You can now use any of the MMSODA visualizations (or MATLAB visualizations for that matter), in just the same way as if the results had been calculated locally.

► 46. Now that you know what information goes where for MMSODA in 'bypass' mode, it may be useful to review some of the other examples in the 'esibayes' directory. On your local machine, explore the configurations for a few of the other 'bypass' mode directories. Try running MMSODA for those configurations, but make sure MATLAB is set to the correct working directory when `mmsoda` is started.

4.3 MMSODA in 'scemua' mode; sequential execution

Now that we have a working example of MMSODA in 'bypass' mode, let's try our hand at something a little more difficult: optimizing the parameters of a dynamic model. This works in more or less the same way as the bypass mode, except that the likelihood is determined by comparing a model prediction to observations, while taking into account the uncertainty of the latter. Let's first make a small digression and look at the general structure of dynamic models. Such models simulate the behavior of a number of states over time. As an example, Fig. 4.3 shows a system in which there are two states. The first state is the water level in a tank which has a small hole in the bottom. The second state is the water level in a tank that has no leaks. The first tank discharges into the second. The rate of discharge Q is dependent on the volume of water in the tank V_{tank_1} and on a resistance parameter R (if there is a big leak in the first tank, the resistance is low, but if the leak is only small, the resistance is high). The model simulates discharge as follows:

$$Q = \frac{V_{tank_1}}{-R} \tag{4.3}$$

Given the state of the upper tank at a given point in time $V_{tank_1}(t)$, Eq. 4.3 may be used to calculate the corresponding flow at time t. Furthermore, the current state $V_{tank_1}(t)$ and flow $Q(t)$ may be used to simulate the state at a later time, $t + \Delta t_{sim}$, provided that Δt_{sim} is sufficiently small:

$$V_{tank_1}(t + \Delta t_{sim}) = V_{tank_1}(t) + Q(t) * \Delta t_{sim} \tag{4.4}$$

$$V_{tank_2}(t + \Delta t_{sim}) = V_{tank_2}(t) - Q(t) * \Delta t_{sim} \tag{4.5}$$

If the initial state of the system $V_{tank_1}(t_0)$ and $V_{tank_2}(t_0)$ is known, and if the resistance parameter R has been assigned, repeated application of Eqs. 4.3–4.5 thus enables constructing a time series of simulated values for V_{tank_1} and V_{tank_2}.

It is quite common that not all the parameter values are known beforehand, so in order to make predictions with the model, it becomes necessary to estimate the model parameter values by comparing simulation results to observations of the system's states. The model must therefore be set up such that it returns the system's state at the times for which an observation is available, otherwise the comparison isn't much use. Since the time interval between observations Δt_{obs} can be (much) larger than the model integration interval Δt_{sim}, multiple model integration steps are often needed in between observation times. Listing 4.1 shows a simple MATLAB script that implements the system depicted in Figure 4.3. A copy of the script has been included as 'other/lintank_script.m'.

Over the next few exercises, we will:

1. prepare a 'constants.mat',
2. prepare a 'conf.mat',
3. prepare the main model m-file by adapting 'lintank_script.m' such that it can be used within the MMSODA framework,
4. construct a likelihood function.

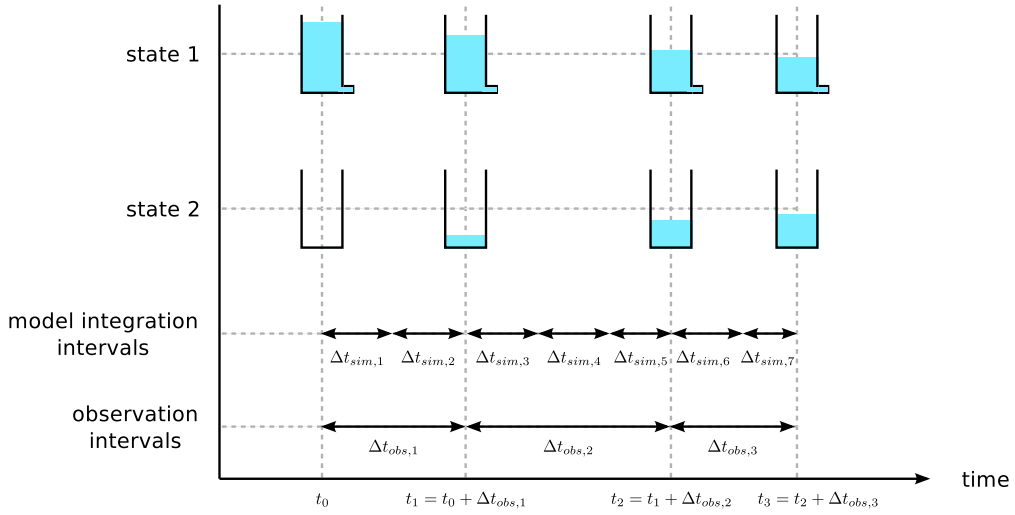


Figure 4.3: The general structure of dynamic models.

► 47. Before we start, make sure you understand how the 'lintank_script.m' works by studying Listing 4.1 and running through it line-by-line using MATLAB's debugging capabilities.

Listing 4.1: Simple MATLAB script that implements the system depicted in Figure 4.3. A copy of this script has been included as 'other/lintank_script.m'.

```
clear        % clear old variables
close all    % close any open figures
clc          % clean the command window

% assign the resistance parameter a value of 100
R = 100;

% initial value of the upper tank
state1Init = 30.0;
% initial value of the lower tank
state2Init = 0;

% initialize 'priorTimes', the array that contains the time of the initial state (0) as well
% as all times at which the model should return a prediction of state1 and state2.
priorTimes = [0,61,147,200];
% initialize the current simulated time as the first entry in 'priorTimes'
tNow = priorTimes(1);
% initialize the last simulated time as the last entry in 'priorTimes'
tEnd = priorTimes(end);

% set the default model integration step
dtSimDefault = 30.5;

% assign the upper tank state variable its initial value
state1 = state1Init;
% assign the lower tank state variable its initial value
state2 = state2Init;

% initialize an array for recording 'tNow', 'state1', and 'state2'
rec = [tNow,state1,state2];

while tNow < tEnd

    % retrieve the index in 'priorTimes' of the next observation time:
    iPriorNext = find(tNow<priorTimes,1,'first');

    % In principle, the model proceeds by time steps of 'dtSimDefault'. However, if that
    % results in values of 'tNow' beyond the next time of observation
    % 'priorTimes(iPriorNext)', then we use a shorter model integration time step
    % 'dtSim = priorTimes(iPriorNext) - tNow' instead.
    dtSim = min([dtSimDefault,priorTimes(iPriorNext) - tNow]);

    % calculate the instantaneous discharge from 'state1' and the resistance parameter 'R':
    Q = state1/-R;

    % integrate the discharge from the first tank over the interval dtSim
    dState = Q * dtSim;

    % decrease the volume in the first tank by dState (which is negative, hence the + sign)
    state1 = state1 + dState;

    % increase the volume in the first tank by dState (which is negative, hence the - sign)
    state2 = state2 - dState;

    % increment the current time by the model integration time step
    tNow = tNow + dtSim;

    % record the current time, state1 and state2
    rec = [rec;tNow,state1,state2];

end

figure(1)
plot(rec(:,1),rec(:,2),'-b.',rec(:,1),rec(:,3),'-m.')
set(gca,'xtick',rec(:,1))
xlabel('Time')
ylabel('State')
legend('state 1','state 2')
```

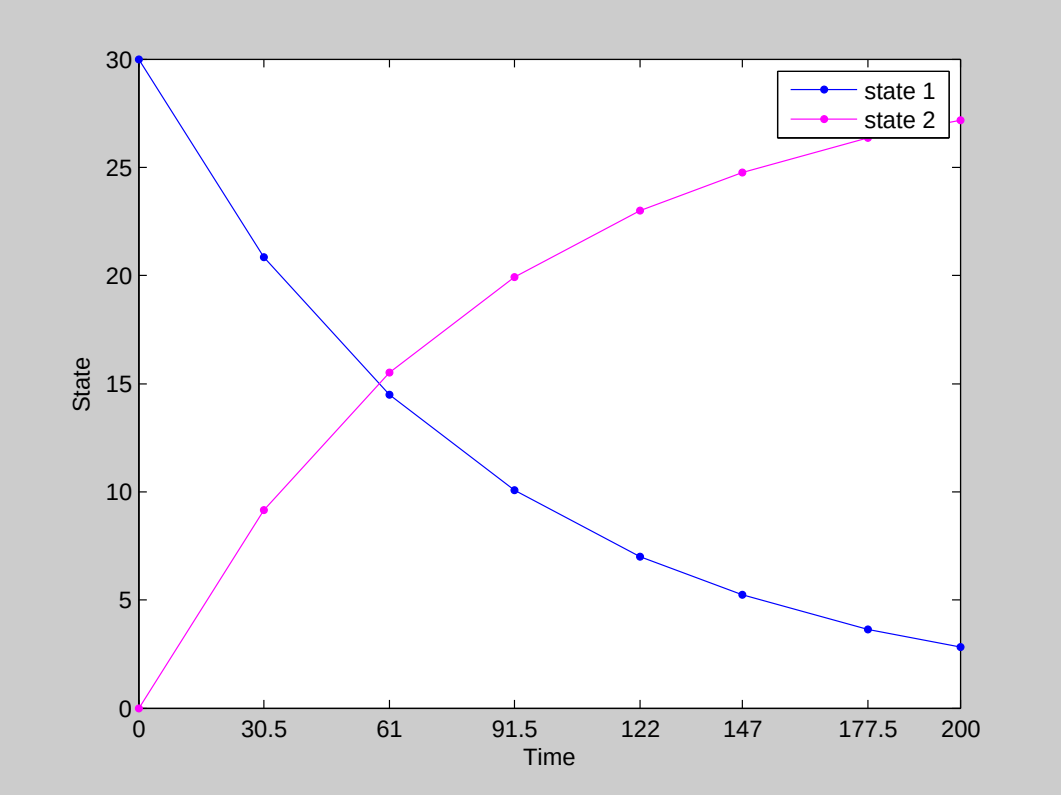


Figure 4.4: Result of running the code from Listing 4.1.

▶ 48. Create a new directory structure with the required subdirectories just like you did before. Call the top directory 'example2'. Verify that the 'example2' directory is at the same level as the 'mmsoda-toolbox' directory.

▶ 49. Read the MMSODA documentation on 'the dynamic model' (see Figure 4.5).

In the initialization part of Listing 4.1, 10 variables (`R`, `state1Init`, `state2Init`, `priorTimes`, `tNow`, `tEnd`, `dtSimDefault`, `rec`, `state1`, and `state2`) are created, but only some of these need to be included in 'constants.mat'. For example, `R` does not need to be in 'constants.mat' because its value will be determined by MMSODA: the variable `R` is assigned by `mmsodaUnpack` (which uses the input argument `parVec` to do so). Similarly, `priorTimes` and its derivatives `tNow` and `tEnd` do not need to be in 'constants.mat' because `priorTimes` is an input argument, too. `state1`, `state2`, and `rec` are all derived from other variables, so they do not need to be part of 'constants.mat' either. Essentially, all we need is `state1Init`, `state2Init`, and `dtSimDefault`.

▶ 50. Write 'makeconstants.m'.

▶ 51. Create a new m-file called 'makeconf.m' just like you did before, but this time make sure that the m-file lists the necessary configuration variables for a 'scemua' optimization. Refer to the configuration variables table in the MATLAB documentation of `mmsoda`, and use the information below to set up the optimization:

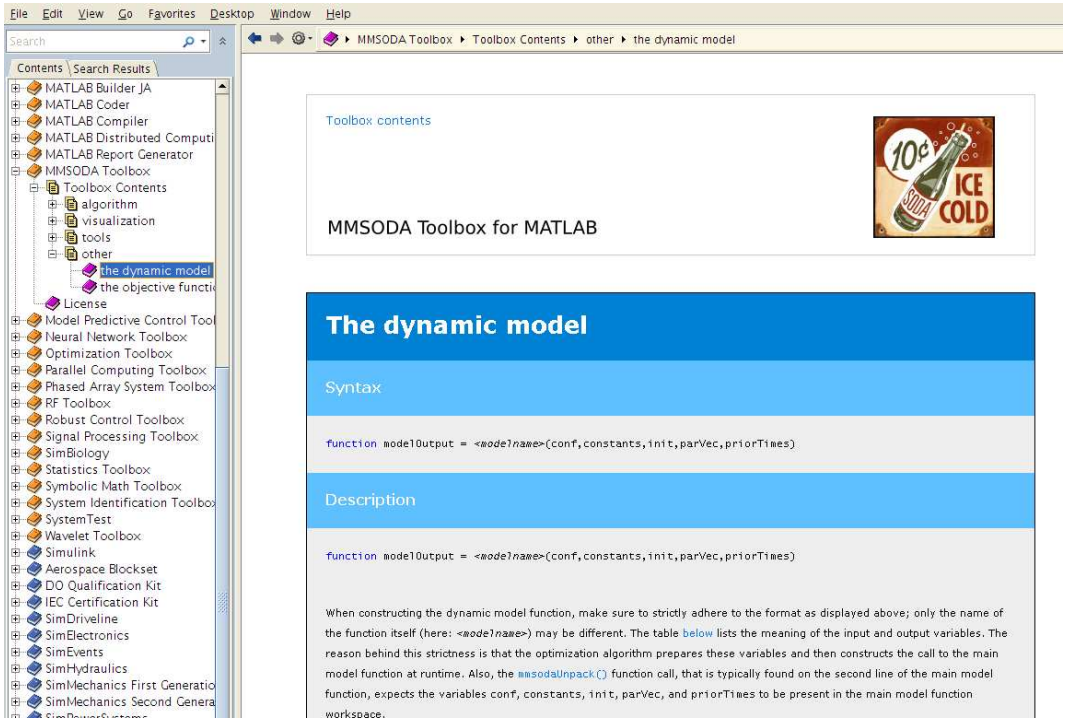


Figure 4.5: MATLAB documentation on how to set up the dynamic model.

1. Set `modeStr` to `'scemua'`;
2. Set `modelName` to `'lintank'`. (We will create 'lintank.m' later);
3. Set `objCallStr` to `'calcLikelihoodState'`. (We will create 'calcLikelihoodState.m' later);
4. Set `parNames` to a cell array of strings with the name of the resistance parameter exactly as used in the dynamic part of the model: `{'R'}`. `R` is the only parameter that will be optimized;
5. For the upper boundary of the parameter space, use 1000.0;
6. For the lower boundary of the parameter space, use 80.0;
7. Fill in the `priorTimes` values by copying from 'lintank_script.m';
8. Set `nOutputs` to 2.

► 52. Copy 'lintank_script.m' to your 'model' subdirectory. Rename it to 'lintank.m'.

► 53. Adapt 'lintank.m' to be a function. Refer to the MMSODA documentation on the dynamic model for the proper way of constructing the function's input and output arguments. Remove all lines from the initialization part that are obsolete given that we want to run it within the MMSODA framework. Afterwards, your code should look like that of Listing 4.2.

Listing 4.2: First 19 lines of 'lintank.m'. You can find a copy of this script at 'other/lintank.m'.

```
1  function modelOutput = lintank(conf,constants,init,parVec,priorTimes)
2
3  % unpack the information from the input arguments and assign it to the correct variables
4  mmsodaUnpack()
5
6  % initialize the current simulated time as the first entry in 'priorTimes'
7  tNow = priorTimes(1);
8
9  % initialize the last simulated time as the last entry in 'priorTimes'
10 tEnd = priorTimes(end);
11
12 % assign the upper tank state variable its initial value
13 state1 = state1Init;
14
15 % assign the lower tank state variable its initial value
16 state2 = state2Init;
17
18 % initialize an array for recording 'tNow', 'state1', and 'state2'
19 rec = [tNow,state1,state2];
```

Next, we need to make sure that the function returns the correct values—we want it to return an array of size `nOutputs x nPrior`. The n^{th} column in the output argument `modelOutput` must contain the values pertaining to the n^{th} time in `priorTimes`. Listing 4.3 shows a simple way of accomplishing this.

Listing 4.3: Last 10 lines of 'lintank.m'. You can find a copy of this script at 'other/lintank.m'.

```
61 % transpose 'rec' because the output argument 'modelOutput' must have 'nOutputs' rows
62 % and nPrior columns
63 rec = rec';
64 % find the columns in the transposed 'rec' array that contain the values of 'state1' and
65 % 'state2' at the times listed in 'priorTimes':
66 c = ismember(rec(1,:),priorTimes);
67 % extract those columns from 'rec' and assign to the output argument 'modelOutput'
68 modelOutput = rec(2:3,c);
69 % overwrite with NaN to comply with what MMSODA expects
70 modelOutput(1:conf.nStatesKF,1) = NaN;
```

▶ 54. Now it's time to test if everything works so far. Make sure MATLAB is in the correct working directory (i.e. one level higher than your 'data', 'model', and 'results' directories). At the MATLAB prompt type:
`>> [evalResults,critGelRub,sequences,metropolisRejects,conf] = mmsoda()`
and press Enter. If all goes well, MMSODA will start printing various messages to your screen. You should see the message '`Evaluating parameter sets 1-100`', and if you haven't commented out the visualization part in 'lintank.m', you should see 100 plots being made before MMSODA crashes with the following error:

```
Error using eval
Undefined function 'calcLikelihoodState' for input arguments of type 'struct'.
>>
```

The error is because MMSODA refers to the configuration variable `objCallStr`, which you set to `'calcLikelihoodState'` earlier on. When MMSODA attempts to run `calcLikelihoodState`, this results in an error because we still have to create 'calcLikelihoodState.m'.

▶ 55. Read the documentation about the objective function (see Figure 4.6).

▶ 56. Create a new m-file in the './model' subdirectory, and call this file 'calcLikelihoodState.m'.

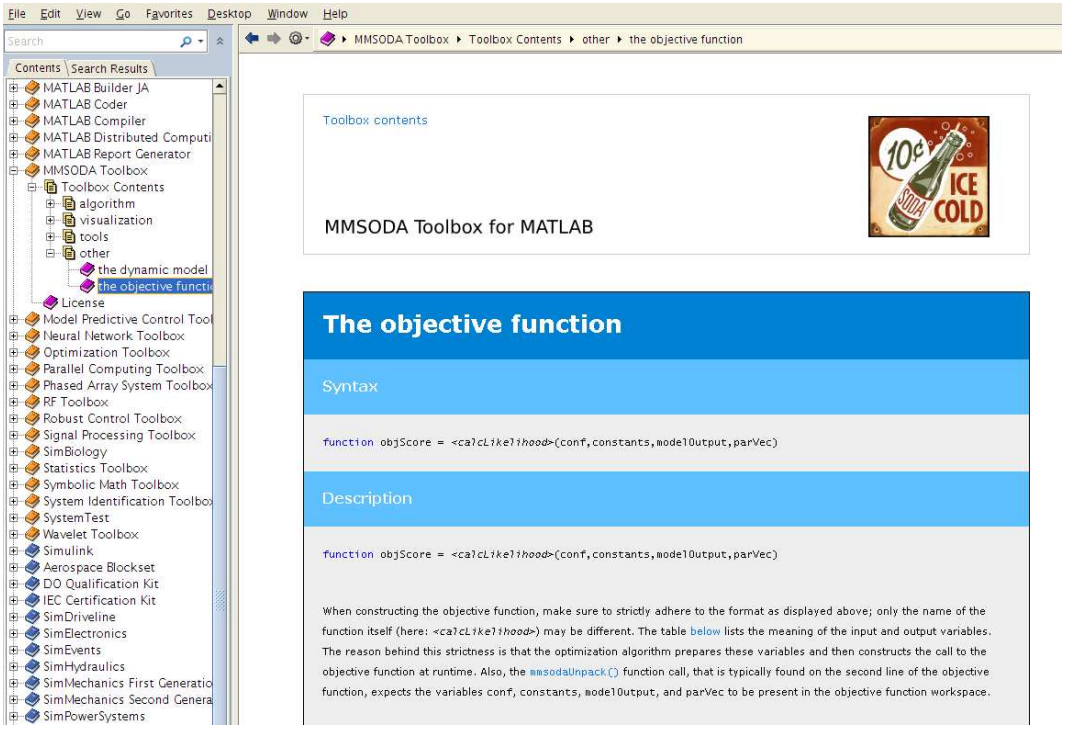


Figure 4.6: MATLAB documentation on how to set up the objective function.

Refer to the documentation and make sure the first line of the objective function is correct.

The objective function we will use is related to the sum of squared residuals or SSR:

$$\mathrm{SSR} = \sum_{t=1}^{n_o} (\hat{x}_t - \tilde{x}_t)^2 \tag{4.6}$$

with $\hat{x}_t$ the t^{th} predicted value of x, $\tilde{x}_t$ the t^{th} observed value of x, and n_o the total number of observations. Note that the number of observations n_o is one less than the number of elements in `conf.nPrior`, since the first column in `model0utput` is not calculated by the model, but instead contains the initial values of the model output variables.

However, we can't use the SSR directly as an objective score, because the SSR is not a log-likelihood (or a probability, for that matter). This problem is easily resolved though, by using the following objective function[1]:

$$\ell = -\frac{1}{2} \cdot n_o \cdot \ln(\mathrm{SSR}) \tag{4.7}$$

[1] Skip forward to Appendix B to see how this was derived.

In order to calculate the SSR, we need some observations. These have been prepared already and are located in the 'other' directory.

▶ 57. Copy 'other/lintank-obs.mat' to your './data' directory.

▶ 58. In 'calcLikelihoodState.m', add
`load('./data/lintank-obs.mat','obs','obsTimes')`

Now that we have the observations in the workspace, we still need to select the corresponding simulations from the `modelOutput` variable. Refer to the documentation on the objective function for a description. We want to select all values pertaining to the first state (since this is the state that we have observations for). We can do so by:

```
% row of interest in 'modelOutput' is #1
r = 1;
nPriors = size(modelOutput,2);
nObs = nPriors-1;

% extract the relevant row from 'modelOutput'
sim = modelOutput(r,1:nPriors);

% Calculate the SSR, but ignore the first entry in 'obs' and 'sim' because those are always
% exactly the same anyway
ssr = sum((obs(1,2:nPriors)-sim(1,2:nPriors)).^2);

% use the SSR to calculate the likelihood
objScore = -(1/2) * nObs * log(ssr);
```

▶ 59. Add the command lines above to your `calcLikelihoodState` in order to let it calculate the SSR and the objective score according to Eq. 4.7.

▶ 60. Make sure you are still in the right working directory. At the MATLAB prompt, type:
`>> [evalResults,critGelRub,sequences,metropolisRejects,conf] = mmsoda()`

Currently, `calcLikelihoodState` loads the observations from file every time it needs to calculate the `objScore`. Since file operations are much slower than memory operations, it is more efficient to load the observations just once (during creation of the constants), and then keep them in the computer's memory.

▶ 61. Cut the `load` statement from `calcLikelihoodState` and paste it into 'makeconstants.m'. Make sure to re-run `makeconstants()`, otherwise './data/constants.mat' will remain unchanged.

▶ 62. Re-read the documentation on MMSODA's `mmsodaUnpack()` function. Go back to 'calcLikelihoodState.m' and use `mmsodaUnpack()` to make the observations available in the objective function's workspace. Restart the optimization to check if everything works.

4.4 MMSODA in 'scemua' mode; parallel execution

▶ 63. Generate a 'Makefile' and a jobscript using `mmsodaPrepParallelFiles()`. Use the same answers as those given on page 28.

▶ 64. Copy the working directory to LISA using WinSCP or an alternative program. Make sure the directory is at the same level as the 'mmsoda-toolbox' directory that should still be

present on the remote system.

► 65. Use PuTTY to start an SSH connection to the LISA cluster.

► 66. Load the necessary software modules and compile your m-files together with the MMSODA m-files.

► 67. Add the executable permission to 'run-mmsoda.sh'.

► 68. Start the optimization on the login node.

So far, we've run our optimizations on one of the login nodes. These nodes are intended for testing small problems and for development work. However, if you want to do some real calculations, then it becomes necessary to submit your job to the PBS queue (as discussed in Chapter 2).

► 69. Submitting your optimization to the PBS scheduler requires a slightly different jobscript. On your local system, run `mmsodaPrepParallelFiles`, but this time indicate that you want to run your optimization as a PBS jobscript, without much verbal feedback, and that you want 00:15:00 walltime on 1 node. We are not interested in saving the timings for the moment. When `mmsodaPrepParallelFiles` finishes, it prints the name of the file it just created ('jobscript-mmsoda.pbs') to the command window. Copy this file to the working directory on the remote system.

► 70. Make sure that PuTTY is in the right directory and type:
`qsub jobscript-mmsoda.pbs`
to add the optimization to the scheduler's queue. Note that it is not necessary to change the permission bits for 'jobscript-mmsoda.pbs' when it is simply an input argument to `qsub`, rather than a program in its own right. Furthermore, it is also not necessary to re-compile your program, since we did not change anything in the code—we're just submitting to a different queue.

► 71. Use the tools discussed in Chapter 2 (e.g. `qstat`, `showq`, `showstart`, `checkjob`) to check on the status of your job.

4.4.1 Standard output and standard error

This is probably as good a time as any to look in more detail at some of the outputs that MMSODA generates. On the remote machine, there should now be a file './results/jobscript-mmsoda.pbs.**o**XXXXXXX' (those X's represent the job id number). This file contains the standard **o**utput of our parallel optimization program. Whenever any of the parallel processes would normally print some message to the command window, on LISA that text will instead end up in the standard output file. The standard output file (and its sister, the standard **e**rror file './results/jobscript-mmsoda.pbs.**e**XXXXXXX') is interesting because it is the first place to look if something is not going like it should.

It is not necessary for you to fully understand what all lines in the standard output mean, but having just a general idea can save you a lot of time and frustration when you run into trouble. Most often, tracking an error is just a matter of spotting the difference between the output that you see when everything is working and when it's not. If nothing else, you

can send the standard output file to one of the LISA administrators, such that (s)he may assist you better in solving your problem.

The standard output files that are generated during MMSODA optimizations typically consist of three parts: first, there is the output that was written when everything was being set up for the MMSODA run; second, there is a middle part that contains output that would be displayed in MATLAB's command window if you were running MMSODA on your local machine. Finally, the third part contains some text that the LISA system adds to every job's standard output. Listings 4.4 and 4.5 show the most interesting parts of the standard output for a single-objective MMSODA optimization in 'scemua' mode. Much of the lines in the first part are generated by bash commands[1] from the jobscript. For example, it includes the ID of the job (lines 3–4), an `ls -l` overview of the files in the current directory (lines 12–22), an overview of the type of CPU in each node (lines 29–45), an overview of the nodes which have been assigned to this job by the scheduler (lines 47–63; the name of each node occurs as many times as there are CPUs in it, so in this case there are 16 'r41n3' entries). Then the necessary modules are loaded and an overview is printed of the available modules (lines 65–77). Next, lines 79–81 add the current directory to the environment variable LD_LIBRARY_PATH to make sure that 'libmmpi.so' can be located by the compiled MATLAB program. The next few lines (83–85) prepare a temporary directory that is needed by the compiled MATLAB program. Lines 87–117 print results of the Linux `ldd` command. `ldd` prints the location of the libraries that are needed by our compiled MATLAB program ('matlabprog') and by the library that we made ('libmmpi.so'). After every `=>` sign in the `ldd` result, there should be an entry; if it says 'not found', something is wrong. For example, if we would not have added the current directory to the LD_LIBRARY_PATH, line 91 would say '`libmmpi.so => not found`'. Next, lines 119–176 constitute a list of so-called 'symbols' that are present inside the 'mmsoda-toolbox/comms/helper.o' file. This file is a C-language object file needed for MPI communication between nodes in the cluster. The next few lines (178–195) show that the MATLAB engine was indeed started 16 times (once for every CPU in the node), and each of those 16 instances of MATLAB recognized immediately that there was no display (since the nodes inside a cluster are primarily set up for calculation, and are therefore not connected to a screen).

The middle part of the standard output is what you normally see in the command window; it starts with a disclaimer message that is generated by the MMSODA code (lines 197–198). Most of the remainder of the middle part consists of 'Evaluating parameter sets X-Y' messages (lines 203–902), indicating the progress of the MMSODA algorithm.

The last part of the standard output file (lines 906–920) is always added by the LISA system. It provides an overview of the resources that were used in executing the job.

[1] See page 14 in Chapter 2.

Listing 4.4: First part of the standard output file. Lines that were too long to fit on the page were wrapped; this is indicated with the line break ↙ and line continuation → symbols.

```
Starting MMSODA job

Result of echo $PBS_JOBID is:
6734963.batch1.lisa.surfsara.nl

Current dir ($PWD) is:
/home/jspaaks
Changing into $PBS_O_WORKDIR
Current dir ($PWD) is:
/home/jspaaks/gitrepo/esibayes/example2

Result of 'ls -l' is:
total 220
drwxrwxr-x 2 jspaaks jspaaks     48 Feb  8 15:05 data
-rw-r--r-- 1 jspaaks jspaaks   2162 Mar 14 10:19 jobscript-mmsoda.pbs
-rwxrwxr-x 1 jspaaks jspaaks 193186 Mar 14 10:20 libmmpi.so
-rw-rw-r-- 1 jspaaks jspaaks    368 Mar  6 13:10 makeconf.m
-rw-rw-r-- 1 jspaaks jspaaks    285 Feb 15 10:34 makeconstants.m
-rw-r--r-- 1 jspaaks jspaaks   1767 Feb 20 13:20 Makefile
-rwxrwxr-x 1 jspaaks jspaaks  11371 Mar 14 10:20 matlabprog
drwxrwxr-x 2 jspaaks jspaaks     50 Feb  1 14:57 model
drwxrwxr-x 2 jspaaks jspaaks     21 Mar 14 10:18 results

Name of the mpirun hostfile is: /scratch/mmsoda-hostfile-G06ssGRP

node r41n3 has 16 CPUs.
A total of 1 batch nodes with a total of 16 CPUs will be used.

Result of 'cat /proc/cpuinfo | grep -e "model name"' on node r41n3:
model name  : Intel(R) Xeon(R) CPU E5-2650L 0 @ 1.80GHz
model name  : Intel(R) Xeon(R) CPU E5-2650L 0 @ 1.80GHz
model name  : Intel(R) Xeon(R) CPU E5-2650L 0 @ 1.80GHz
model name  : Intel(R) Xeon(R) CPU E5-2650L 0 @ 1.80GHz
model name  : Intel(R) Xeon(R) CPU E5-2650L 0 @ 1.80GHz
model name  : Intel(R) Xeon(R) CPU E5-2650L 0 @ 1.80GHz
model name  : Intel(R) Xeon(R) CPU E5-2650L 0 @ 1.80GHz
model name  : Intel(R) Xeon(R) CPU E5-2650L 0 @ 1.80GHz
model name  : Intel(R) Xeon(R) CPU E5-2650L 0 @ 1.80GHz
model name  : Intel(R) Xeon(R) CPU E5-2650L 0 @ 1.80GHz
model name  : Intel(R) Xeon(R) CPU E5-2650L 0 @ 1.80GHz
model name  : Intel(R) Xeon(R) CPU E5-2650L 0 @ 1.80GHz
model name  : Intel(R) Xeon(R) CPU E5-2650L 0 @ 1.80GHz
model name  : Intel(R) Xeon(R) CPU E5-2650L 0 @ 1.80GHz
model name  : Intel(R) Xeon(R) CPU E5-2650L 0 @ 1.80GHz
model name  : Intel(R) Xeon(R) CPU E5-2650L 0 @ 1.80GHz

Result of 'cat /scratch/mmsoda-hostfile-G06ssGRP' is:
r41n3
r41n3
r41n3
r41n3
r41n3
r41n3
r41n3
r41n3
r41n3
r41n3
r41n3
r41n3
r41n3
r41n3
r41n3
r41n3

Unloading module matlab
Loading module openmpi/gnu
Message from module openmpi/gnu/1.6.3:
This module is now default. If you encounter problems:
recompile or use the 1.4.5 module.
```

```
More info:
https://www.sara.nl/systems/lisa/news/2012-11-02-ompi1.4-ompi1.6
Loading module mcr
Result of 'module list 2>&1' is:
Currently Loaded Modulefiles:
 1) libgfortran/32/1   4) licenses/default   7) openmpi/gnu/1.6.3
 2) stdenv/1.1         5) sara/1             8) mcr/64/v718
 3) compilerwrappers   6) ofed/64/1.5.2

Adding the current workdir to the library path environment variable
The current LD_LIBRARY_PATH is:
/sara/sw/mcr/64/v718/v80/runtime/glnxa64:/sara/sw/mcr/64/v718/v80/bin/glnxa64:/sara/
→ sw/mcr/64/v718/v80/sys/os/glnxa64:/sara/sw/mcr/64/v718/v80/sys/java/jre/glnxa64/
→ jre/lib/amd64/native_threads:/sara/sw/mcr/64/v718/v80/sys/java/jre/glnxa64/jre/lib
→ /amd64/server:/sara/sw/mcr/64/v718/v80/sys/java/jre/glnxa64/jre/lib/amd64:/sara/sw
→ /openmpi-gnu-1.6.3/lib:/sara/sw/ofed/1.5.2/64/lib64:/sara/sw/libgfortran/32/1/lib
→ :/home/jspaaks/gitrepo/esibayes/example2

Setting MCR_CACHE_ROOT to: /scratch/mmsoda-Gt1ygdF7
The current MCR_CACHE_ROOT is:
/scratch/mmsoda-Gt1ygdF7

Result of 'ldd matlabprog' is:
    linux-vdso.so.1 =>  (0x00007fff5c3ff000)
    libmwmclmcrrt.so.8.0 => /sara/sw/mcr/64/v718/v80/runtime/glnxa64/libmwmclmcrrt.so
→ .8.0 (0x00002b24dcd44000)
    libm.so.6 => /lib/libm.so.6 (0x00002b24dcfac000)
    libmmpi.so => /home/jspaaks/gitrepo/esibayes/example2/libmmpi.so (0
→ x00002b24dd22e000)
    libmpi.so.1 => /sara/sw/openmpi-gnu-1.6.3/lib/libmpi.so.1 (0x00002b24dd435000)
    libpthread.so.0 => /lib/libpthread.so.0 (0x00002b24dd9df000)
    libc.so.6 => /lib/libc.so.6 (0x00002b24ddbfb000)
    libmwcpp11compat.so => /sara/sw/mcr/64/v718/v80/runtime/glnxa64/../../bin/glnxa64
→ /libmwcpp11compat.so (0x00002b24ddf5e000)
    libdl.so.2 => /lib/libdl.so.2 (0x00002b24de167000)
    libstdc++.so.6 => /sara/sw/mcr/64/v718/v80/runtime/glnxa64/../../sys/os/glnxa64/
→ libstdc++.so.6 (0x00002b24de36b000)
    libgcc_s.so.1 => /sara/sw/mcr/64/v718/v80/runtime/glnxa64/../../sys/os/glnxa64/
→ libgcc_s.so.1 (0x00002b24de67b000)
    libibverbs.so.1 => /sara/sw/ofed/1.5.2/64/lib64/libibverbs.so.1 (0
→ x00002b24de891000)
    libtorque.so.2 => /usr/lib/libtorque.so.2 (0x00002b24dea9e000)
    libnuma.so.1 => /usr/lib/libnuma.so.1 (0x00002b24dedb6000)
    librt.so.1 => /lib/librt.so.1 (0x00002b24defbe000)
    libnsl.so.1 => /lib/libnsl.so.1 (0x00002b24df1c6000)
    libutil.so.1 => /lib/libutil.so.1 (0x00002b24df3df000)
    /lib64/ld-linux-x86-64.so.2 (0x00002b24dcb24000)

Result of 'ldd libmmpi.so' is:
    linux-vdso.so.1 =>  (0x00002b1b8b1a5000)
    libmwmclmcrrt.so.8.0 => /sara/sw/mcr/64/v718/v80/runtime/glnxa64/libmwmclmcrrt.so
→ .8.0 (0x00002b1b8b5b0000)
    libm.so.6 => /lib/libm.so.6 (0x00002b1b8b818000)
    libpthread.so.0 => /lib/libpthread.so.0 (0x00002b1b8ba9a000)
    libc.so.6 => /lib/libc.so.6 (0x00002b1b8bcb6000)
    libmwcpp11compat.so => /sara/sw/mcr/64/v718/v80/runtime/glnxa64/../../bin/glnxa64
→ /libmwcpp11compat.so (0x00002b1b8c019000)
    libdl.so.2 => /lib/libdl.so.2 (0x00002b1b8c222000)
    libstdc++.so.6 => /sara/sw/mcr/64/v718/v80/runtime/glnxa64/../../sys/os/glnxa64/
→ libstdc++.so.6 (0x00002b1b8c426000)
    libgcc_s.so.1 => /sara/sw/mcr/64/v718/v80/runtime/glnxa64/../../sys/os/glnxa64/
→ libgcc_s.so.1 (0x00002b1b8c736000)
    /lib64/ld-linux-x86-64.so.2 (0x00002b1b8b187000)

Result of 'nm ../mmsoda-toolbox/comms/helper.o' is:
0000000000000000 r .LC0
0000000000000007 r .LC1
0000000000000000 r .LC10
0000000000000062 r .LC11
0000000000000064 r .LC12
000000000000006b r .LC13
0000000000000089 r .LC14
0000000000000020 r .LC15
```

```
0000000000000008e r .LC16
00000000000000ab r .LC17
0000000000000040 r .LC18
0000000000000060 r .LC19
000000000000000c r .LC2
00000000000000ae r .LC20
00000000000000b0 r .LC21
0000000000000027 r .LC3
000000000000002f r .LC4
0000000000000039 r .LC5
000000000000003d r .LC6
0000000000000000 r .LC7
0000000000000057 r .LC8
000000000000005d r .LC9
0000000000000580 T DeserializeVar
0000000000000310 T SerializeVar
0000000000000000 T SetTimeStamp
                 U _GLOBAL_OFFSET_TABLE_
                 U fclose
                 U fopen
                 U fread
                 U fseek
                 U ftell
                 U fwrite
                 U getenv
                 U malloc
                 U matClose
                 U matGetVariable
                 U matOpen
                 U matPutVariable
                 U memcpy
                 U mexCallMATLAB
                 U mexErrMsgTxt
                 U mexGetVariable
                 U mexPutVariable
                 U mexWarnMsgTxt
                 U mktemp
                 U mxAddField
                 U mxCreateDoubleMatrix
                 U mxCreateNumericMatrix
                 U mxDestroyArray
                 U mxGetField
                 U mxGetFieldNumber
                 U mxGetNumberOfElements
                 U mxGetPr
                 U mxRemoveField
                 U mxSetFieldByNumber
                 U remove
                 U strcpy

Starting mpirun
Setting verbosity to level 0.
Warning: No display specified.  You will not be able to display graphics on the ↙
→ screen.
Warning: No display specified.  You will not be able to display graphics on the ↙
→ screen.
Warning: No display specified.  You will not be able to display graphics on the ↙
→ screen.
Warning: No display specified.  You will not be able to display graphics on the ↙
→ screen.
Warning: No display specified.  You will not be able to display graphics on the ↙
→ screen.
Warning: No display specified.  You will not be able to display graphics on the ↙
→ screen.
Warning: No display specified.  You will not be able to display graphics on the ↙
→ screen.
Warning: No display specified.  You will not be able to display graphics on the ↙
→ screen.
Warning: No display specified.  You will not be able to display graphics on the ↙
→ screen.
Warning: No display specified.  You will not be able to display graphics on the ↙
```

```
    → screen.
191 Warning: No display specified.  You will not be able to display graphics on the ↙
    → screen.
192 Warning: No display specified.  You will not be able to display graphics on the ↙
    → screen.
193 Warning: No display specified.  You will not be able to display graphics on the ↙
    → screen.
194 Warning: No display specified.  You will not be able to display graphics on the ↙
    → screen.
195 Warning: No display specified.  You will not be able to display graphics on the ↙
    → screen.
196
197 % This is a pre-alpha release of the MMSODA simultaneous
198 % parameter optimization and data assimilation software.
199
200 Setting value of 'conf.nMembers' to 1.
201
202 SCEMUA-SO run started on: March 14, 2013 10:22:15
203 +00s: Evaluating parameter sets 1-100
204 +00s: Evaluating parameter sets 101-120
205 +01s: Evaluating parameter sets 121-140
206 +01s: Evaluating parameter sets 141-160
207 +01s: Evaluating parameter sets 161-180
```

Listing 4.5: Last part of the standard output file. Lines that were too long to fit on the page were wrapped; this is indicated with the line break ↙ and line continuation → symbols.

```
898 +44s: Evaluating parameter sets 13981-14000
899 +44s: Evaluating parameter sets 14001-14020
900 +44s: Evaluating parameter sets 14021-14040
901 +44s: Evaluating parameter sets 14041-14060
902 +44s: Evaluating parameter sets 14061-14080
903 Convergence achieved.
904 SCEMUA-SO run completed on: March 14, 2013 10:22:59
905 The run took 0d 0h 0m 44s.
906 Q: "...sara_stats": --------------------Begin of SARA epilogue ↙
    → ---------------------------------
907 Q: "...sara_stats": This output was generated by the SARA epilogue script
908 Q: "...sara_stats": Your job [jobscript-mmsoda.pbs] has been executed in queue [ ↙
    → express] with
909 Q: "...sara_stats": the following PBS arguments:
910 Q: "...sara_stats":    ncpus=1,neednodes=1,nodes=1,walltime=00:03:00
911 Q: "...sara_stats":    .
912 Q: "...sara_stats": Resources used in job [6734963.batch1.lisa.surfsara.nl] with name ↙
    →  [jobscript-mmsoda.pbs]:
913 Q: "...sara_stats":    cput=00:13:12,mem=2217356kb,vmem=17188164kb,walltime=00:01:00
914 Q: "...sara_stats":    r41n3
915 Q: "...sara_stats":    .
916 Q: "...sara_stats": Job start and end time:
917 Q: "...sara_stats":   Job start time: Thu Mar 14 10:22:02 CET 2013
918 Q: "...sara_stats":   Job end time  : Thu Mar 14 10:23:04 CET 2013
919 Q: "...sara_stats": ----------------------End of SARA epilogue ↙
    → ---------------------------------
920 I: Last 14 quoted lines were generated by promiser "/var/spool/torque/mom_priv/ ↙
    →  epilogue.d/sara_stats"
```

4.4.2 Analyzing the parallelization overhead

When dealing with parallel programs, it is often useful to monitor how much time is lost in parallelization overhead in relation to the CPU time (recall Fig. 3.1). The MMSODA Toolbox for MATLAB enables you to record and visualize the timings of all relevant events (send-

ing and receiving data, serializing[1] and deserializing, waiting for input, etc.) on all nodes that have been assigned to the job. Recording the timings does slow the optimization down slightly, so the default behavior is not to record any events. However, enabling the time-keeping functionality is simply a matter of re-running the `mmsodaPrepParallelFiles()` function, and answering 'a : yes' to the last question 'Would you like to save the timings?'.

▶ 72. Re-run `mmsodaPrepParallelFiles()` and enable time-keeping. Copy the newly created jobscript to the cluster. Submit the new jobscript using `qsub`.

▶ 73. After the job finishes, copy the contents of the './results' directory from the remote system to your local system.

▶ 74. Read the documentation on `mmsodaAnalyzeTimings` and visualize the timings for the linear tank example. Would you say that parallel optimization of the linear tank model is a coarse-grained or a fine-grained problem?

At this point, you are familiar with MMSODA in 'bypass' mode and in 'scemua' mode, and you know how to run MMSODA optimizations locally as well as on the LISA cluster. In the next few sections, we will cover the 'reset' mode and 'soda' modes, but the important point is that the workflow as you know it from the previous projects (as summarized in Fig. 4.7) is the same for all MMSODA projects.

▶ 75. Now that you know what information goes where for MMSODA in 'scemua' mode, it may be useful to review some of the other examples in the 'esibayes' directory. On your local machine, explore the configurations for a few of the other 'scemua' mode directories. Run MMSODA for those configurations, but make sure MATLAB is set to the correct working directory when `mmsoda` is started.

4.5 MMSODA in 'reset' mode; sequential execution

▶ 76. On your local system, make a copy of the 'example2' directory, and rename the copy to 'example3'. Clean up the 'example3' directory by removing 'matlabprog', 'libmmpi.so', 'Makefile', 'jobscript-mmsoda.pbs' as well as any results pertaining to the 'example2' case. Set your MATLAB working directory to the newly created 'example3' directory.

▶ 77. Remove the 'constants.mat' and 'lintank-obs.mat' files from the 'example3/data' directory. Copy 'lintank-obs-unobserved-input.mat' from the 'other' directory to 'example3/data'.

▶ 78. On the MATLAB command line, load the `obsTimes` and `obs` variables from the './data/lintank-obs-unobserved-input.mat' file by typing:

```
>> load('./data/lintank-obs-unobserved-input.mat','obsTimes','obs')
```

[1] Serialization is the process of collecting all the arrays that need to be sent over to a different machine, such that the arrays occupy a consecutive part of the computer's memory. This part of the memory can then be sent to another machine.

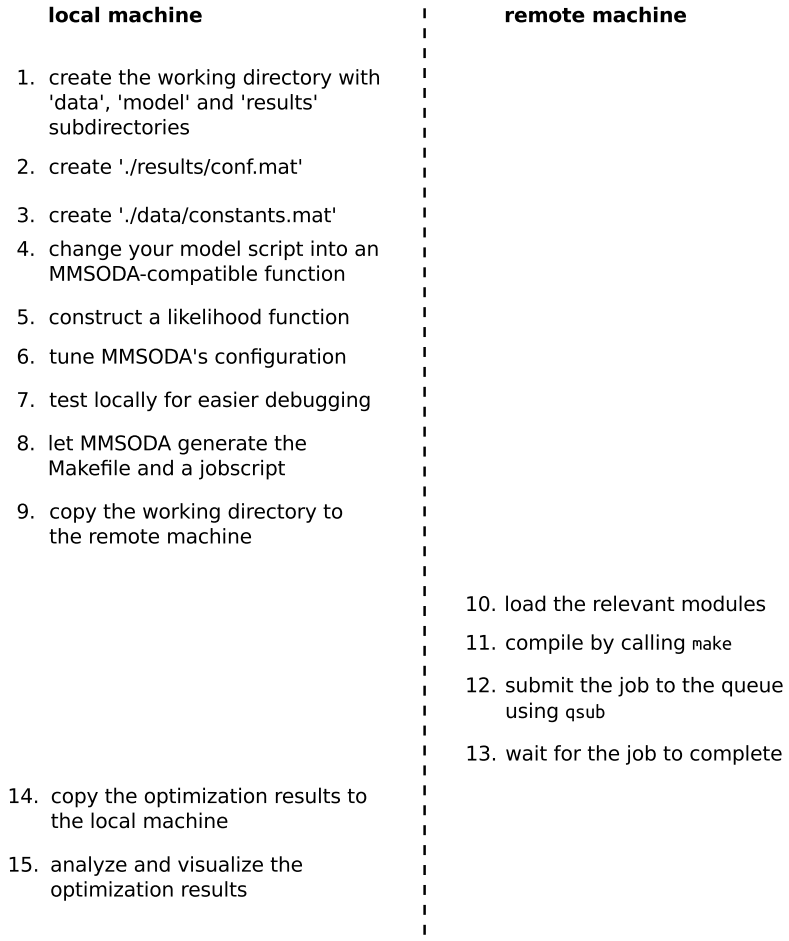


Figure 4.7: Typical workflow for setting up MMSODA optimizations. Note that the items under 'remote machine' are executed in a terminal program such as PuTTY.

► 79. Visualize the data that were just loaded by:
`>> plot(obsTimes,obs(1,:),'-b.')`
Your figure should be similar to Fig. 4.8.

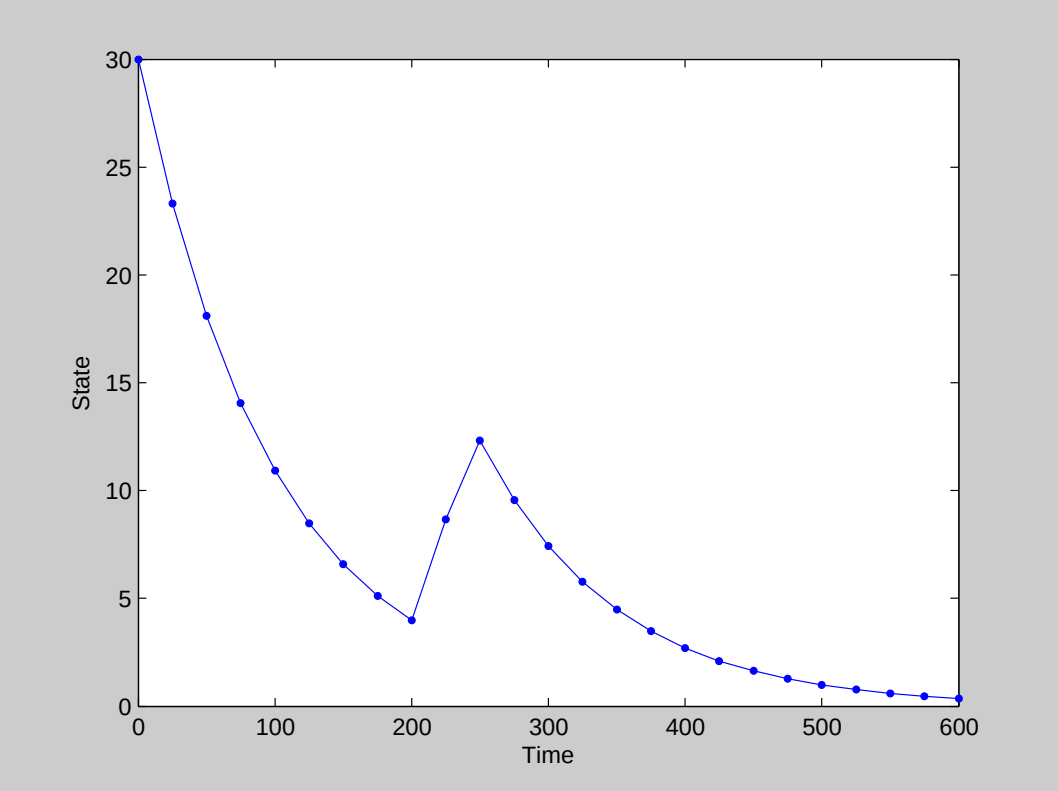


Figure 4.8: Simple `plot` of the data in './data/lintank-obs-unobserved-input.mat'. The true value of the resistance parameter in this figure is 100.

From Fig. 4.8, it is clear that something happened around Time=200–250. It seems that water was added to the upper tank—that is to say, the upper tank has been subjected to an unobserved input. This kind of error is very common in hydrology; for example if the linear tank simulates the streamflow from a particular catchment, it can sometimes happen that streamflow in the river increases (as in Fig. 4.8), without any rain being recorded during the preceding days. Most often, these kinds of apparent impossibilities are explained by a rainstorm which did indeed pass over (some of) the catchment, but which missed the location of the rain gauge. Observational error like that can play havoc with the effectiveness of the calibration, as we shall see shortly.

In this section, we will look at what happens if we take a naive approach to calibrating the lintank model to the data of Fig. 4.8. For this, we need to make a few changes in the MMSODA configuration. For example, we need the model to generate predictions at `priorTimes = [0:25:600]`, and we want to use the data from './data/lintank-obs-unobserved-input.mat' in the objective function.

► 80. Make the necessary changes to 'makeconf.m' and 'makeconstants.m' to implement these changes. Also change the value of the `dtSimDefault` model constant to 2.0 and set the lower and upper boundaries of the parameter space to 10 and 500, respectively. Furthermore, set the `doPlot` configuration variable to `true`, such that MMSODA will produce some standardized visualizations as the optimization progresses. Finally, set `nModelEvalsMax` to `7000`.

► 81. Run the updated versions of 'makeconf.m' and 'makeconstants.m' and verify that the necessary files are created. Then, start the SCEM-UA optimization in the regular way from the MATLAB command window.

Listing 4.6 on page 49 demonstrates how parameter uncertainty intervals can be constructed using the information in `evalResults`. A copy of the listing has been included as 'calcUncertaintyIntervals.m' in the './other' directory.

► 82. Study the script and use it to visualize the model prediction for the best parameter set, along with the parameter uncertainty intervals for the case of Fig. 4.8.

Listing 4.6: Simple MATLAB script that demonstrates how parameter uncertainty intervals can be constructed from the information in `evalResults`. A copy of this script has been included as 'other/-calcUncertaintyIntervals.m'.

```
% load the MMSODA configuration and the 'evalResults' array
load('./results/scemua-so-results.mat','conf','evalResults')

% load the model constants from the data subdirectory
constants = load('./data/constants.mat');

% set the initial state to empty for scemua run (nStatesKF = 0);
init = [];

% define the number of model outputs to include in the construction of the
% parameter uncertainty intervals:
nIntervals = 250;

% count the number of rows in 'evalResults'
nRows = size(evalResults,1);

% pre-allocate the array that will hold the model outputs
allModelOutputs = repmat(NaN,[nIntervals,conf.nPrior]);

% iterate over the last 'nIntervals' rows in 'evalResults', retrieve the
% parameter set, re-run the model structure, and store the model output
for k = nRows+(-nIntervals+1:1:0)

    % retrieve the k-th parameter set
    parVec = evalResults(k,conf.parCols);

    % % re-run the model with the k-th parameter set
    modelOutput = lintank(conf,constants,init,parVec,conf.priorTimes);

    % store the model output
    allModelOutputs(k+nIntervals-nRows,1:conf.nPrior) = modelOutput;

end
% calculate the 2.5, 50, and 97.5 percent parameter uncertainty percentiles
% using MMSODA's 'mmsodaPrctile' function:
percentiles = mmsodaPrctile(allModelOutputs,[2.5,50,97.5]);

% retrieve a list of unique parameter combinations that have the best
% objective scores using MMSODA's 'mmsodaBestParsets' function
bestsets = mmsodaBestParsets(conf,evalResults,'unique');

% pre-allocate the array that holds the model outputs associated with the
% best objective scores
bestModelOutputs = repmat(NaN,[size(bestsets,1),conf.nPrior]);

% use the same principle as before to iterate over the rows in the list of
% best parameter sets, re-run the model, and store the output
for k = 1:size(bestsets,1)

    parVec = bestsets(k,1:conf.nOptPars);
    modelOutput = lintank(conf,constants,init,parVec,conf.priorTimes);
    bestModelOutputs(k,1:conf.nPrior) = modelOutput;

end

% plot the observations along with the 2.5% parameter uncertainty
% percentile, the median, the 97.5% parameter uncertainty percentile, and
% the best model output associated with the best parameter combination
figure
h = plot(constants.obsTimes,constants.obs,'om-',...
    conf.priorTimes,percentiles(1,1:conf.nPrior),'--k',...
    conf.priorTimes,percentiles(2,1:conf.nPrior),'-k',...
    conf.priorTimes,percentiles(3,1:conf.nPrior),'--k',...
    conf.priorTimes,bestModelOutputs,'-b.');
xlabel('Time')
ylabel('State')
legend([h([1,2,3,5])],'obs','95% parameter uncetainty interval','median','best sim')
```

As you can see from the resulting figure, optimizing the model without allowing for the unobserved input to the upper tank can lead to biased parameter estimates—even though the best parameter set minimizes the residuals, the corresponding model output is plainly wrong. In the rest of this section, we will look at data assimilation. Data assimilation provides a way of dealing with uncertain model states (for example as a result of uncertain model forcings, initial state or model structure error). MMSODA supports two types of data assimilation: in the first, full confidence is placed on the observation, i.e. whenever the simulated time reaches a time at which an observation is available, the model state pertaining to that time (known as *prior* state or *forecast* state) is abandoned, and the value of the model state is reset to the value of the observation. In MMSODA parlance, this mode is known as the 'reset' mode. The second type of data assimilation that MMSODA supports is the 'soda' mode, which we will study in greater detail in section 4.6.

According to `mmsoda`'s documentation, the following configuration variables are required for running MMSODA in 'reset' mode when using one objective (we do not use `initMethodNOKF`, `initValuesNOKF` and `namesNOKF` since we are only interested in the model state for the moment and do not bother with any additional variables):

1. `initMethodKF`
2. `initValuesKF`
3. `modeStr`
4. `modelName`
5. `objCallStr`
6. `obsState`
7. `parNames`
8. `parSpaceHiBound`
9. `parSpaceLoBound`
10. `priorTimes`
11. `stateNamesKF`
12. `stateSpaceHiBound`
13. `stateSpaceLoBound`

As you can see, you are already familiar with most of these, but there are a few new ones as well. `initMethodKF` and `initValuesKF` always go together. Their purpose is to initialize the correct initial values to those state variables which are subject to state updating. For our case, that means that `initMethodKF` must be set to `'reference'` (this is currently the only supported method), and that `initValuesKF` must be set to `30.0` (the initial value of the `state1` variable). This implies that `state1Init` no longer needs to be part of the model constants, and that `state1` no longer needs to be initialized by assigning `state1Init` to `state1` in 'lintank.m'.

Configuration variable `obsState` is used to tell MMSODA what the observed values of the state are for every time in `priorTimes`. For our case, that means that `obsState` must be assigned the values from variable `obs` from './data/lintank-unobserved-input.mat'.

You also have to indicate which model variables you want to be part of the Ensemble Kalman Filter scheme (Note that we are using the simplest scheme here, i.e. the 'reset' scheme). This is done through the `stateNamesKF` variable, which is simply a list of variable names as they are used in the model. We only have one variable, so you need to set `stateNamesKF` to `{'state1'}`.

Finally, you need to specify the limits on the state space, i.e. for each state, you define what the maximum and minimum values are, using `stateSpaceHiBound` and `stateSpaceLoBound`, respectively. For this data set, reasonable values are 40.0 and 0.0, respectively.

▶ 83. Make the necessary changes to 'makeconf.m', 'makeconstants.m', and 'lintank.m'. Start the 'reset' mode optimization from the command line like you did before. MMSODA will most likely not run just yet; a couple of things need to be resolved first. Interpret the error messages and make the required changes until MMSODA will start without any errors.

▶ 84. Once the optimization completes after 7000 model evaluations, use 'calcUncertaintyIntervals.m' to visualize the parameter uncertainty (but change the name of the *.mat file from 'scemua-so-results.mat' to 'reset-so-results.mat'). If the result is not quite what you hoped it would be; don't worry, this is just because 'calcUncertaintyIntervals.m' does not reset the values of the prior states when constructing the uncertainty intervals. Luckily, there is an easy way around this problem.

▶ 85. Read the description of the `saveEnKFResults` and `startFromUniform` configuration variables in `mmsoda`'s documentation.

▶ 86. Change 'makeconf.m' such that it will evaluate an additional 500 parameter combinations. Make sure to set `startFromUniform` to `false` (we want to resume the previous run) and set `saveEnKFResults` to `true`, to be able to plot uncertainty intervals for the prior state values later on. Re-run 'makeconf.m' and start the optimization as usual.

▶ 87. When the additional 500 parameter sets have been evaluated, verify that you have a bunch of new files in the './results' directory, whose names are formatted like 'reset-so-results-enkf-evals-Z-Z.mat', with Z representing an integer number.

▶ 88. Read the documentation on `mmsodaPlotEnsemble`. Make a new figure using `mmsodaSubplotScreen(2,2,1)` and visualize the prior state values associated with the last parameter combination.

▶ 89. Once that works, try to visualize the last 20 parameter sets using `mmsodaPlotEnsemble`'s optional parameters.

As you can see, there are quite a few parameter combinations that do not result in a very good fit. It would seem then that the uncertainty intervals associated with these model outputs are quite large. This is not the case, though; it is simply because the default behavior for `mmsodaPlotEnsemble` is to include all model outputs in the figure, regardless of whether they were accepted or rejected by the Metropolis part of MMSODA. If you want to view only the model outputs that were accepted by the Metropolis part of MMSODA, you should use `mmsodaPlotEnsemble`'s `'replaceRejected'` optional parameter.

▶ 90. Make a new figure using `mmsodaSubplotScreen(2,2,2)`. Visualize the same selection of parameter combinations as you did previously, but this time, let `mmsodaPlotEnsemble` replace the rejected outputs with model outputs that were accepted. Note that the model evaluation number in the figure's title is no longer simply `[7481:7500]` but instead shows the model evaluation numbers that replaced the Metropolis-rejected ones.

▶ 91. Make a new figure using `mmsodaSubplotScreen(2,1,2)`. Use `mmsodaMatrixOfScatter` to plot the evaluation number on the x-axis against the parameter value on the y-axis for the

last part of the record (see optional parameter `'nHistory'`). Use the resulting figure to verify that `mmsodaPlotEnsemble` replaced the correct model outputs.

Now that we have some idea about what sort of model outputs are probable, let's calculate the 95% parameter uncertainty intervals. MMSODA comes prepackaged with a function that retrieves the correct model outputs from the relevant files in './results' and that calculates time series of arbitrary percentiles, while taking into account the intermediate state updating that has been performed by MMSODA.

▶ 92. Read the MMSODA documentation on `mmsodaCalcUncertInts` and use it to calculate the 2.5%, 50% and 97.5% percentiles. The code snippet in Listing 4.7 may serve as an example of how `mmsodaCalcUncertInts`'s result may be used, while Fig. 4.9 shows what the result looks like.

Listing 4.7: MATLAB code snippet for visualizing parameter uncertainty intervals for the lintank model. A copy of this script has been included as 'other/patch_uncertainty_snippet.m'.

```
% load variables 'conf','evalResults', 'sequences', and 'metropolisRejects'
% from the reset-mode optimization results:
load('./results/reset-so-results.mat','conf','evalResults',...
    'sequences','metropolisRejects')

% load the observations from file:
constants = load('./data/constants');

% define the percentiles:
prc = [2.5,50,97.5];

% define the number of model outputs to include in the construction of the
% parameter uncertainty intervals:
nIntervals = 250;

% count the number of rows in 'evalResults':
nRows = size(evalResults,1);

% calculate the evaluation numbers of the parameter combinations whose
% associated output are used in constructing the uncertainty intervals:
evalNumbers = nRows + (-nIntervals+1:1:0);

% use the 'mmsodaCalcUncertInts' function to retrieve the correct
% data and to calculate precentiles:
[x,y] = mmsodaCalcUncertInts(conf,evalResults,metropolisRejects,prc,evalNumbers);

% compose the x and y data for the patch object:
xp = [x(1,2:2*conf.nPrior),fliplr(x(1,2:2*conf.nPrior)),x(1,2)];
yp = [y(1,2:2*conf.nPrior),fliplr(y(3,2:2*conf.nPrior)),y(1,2)];

% make the figure:
mmsodaSubplotScreen(2,2,2)
clf
h1 = patch(xp,yp,'k');
hold on
h2 = plot(x,y(2,:),'-k');
h3 = plot(constants.obsTimes,constants.obs,'sm');
set(h1,'facecolor',0.5*[1,1,1],'edgecolor',0.5*[1,1,1])
set(h3,'markerfacecolor','none','markersize',8)
xlabel('Time')
ylabel('State')
box on
legend([h1,h2,h3],'95% parameter uncertainty interval','median','obs')
```

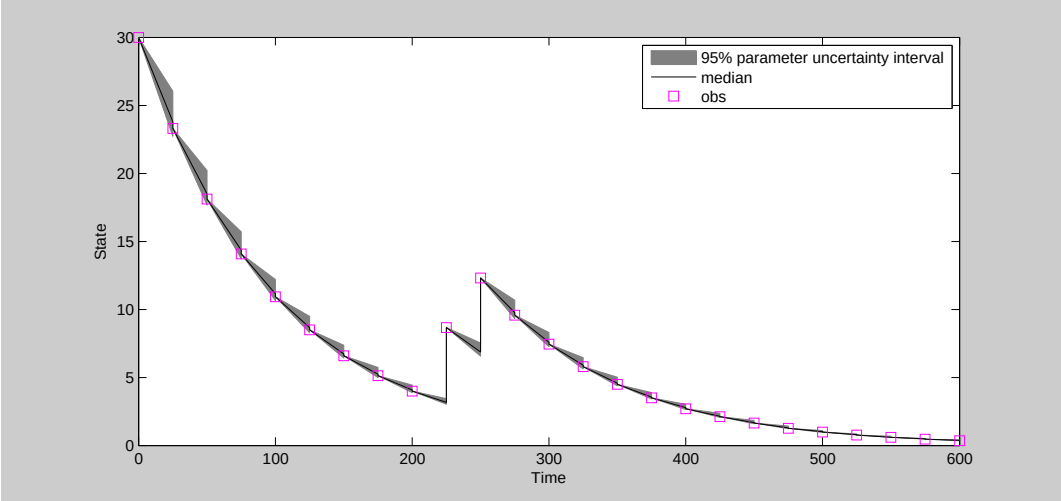


Figure 4.9: Result of running the code from Listing 4.7.

4.6 MMSODA in 'soda' mode; sequential execution

▶ 93. On your local machine, copy the 'example3' directory to a new directory, 'example4'. Clean up the './results' subdirectory, and remove 'matlabprog', 'libmmpi.so', and 'Makefile', as well as any jobscripts pertaining to 'example3'.

The 'soda' mode is the most complex and most compute-intensive mode. It is similar to the 'reset' mode in that it updates the model states during the simulation. The way the updating is performed, however, is different. The difference between 'scemua', 'reset', and 'soda' modes essentially comes down to how much confidence is placed on the observed values of the state in comparison to how much confidence is placed on the simulated values of the state. For example, in 'scemua' mode, all of the confidence is placed on the simulated values—the model can be run from the time of the initial state (`conf.priorTimes(1)`) to the time at which the last prediction is required (`conf.priorTimes(end)`), without the need for temporarily halting the model at `conf.priorTimes(2:end-1)` to do any state updating. In contrast to this, the 'reset' mode places full condidence on the observations: at the times when an observation is available, the model is temporarily halted, and its states are reset to the value of the observed state. The 'soda' mode then, provides sort of a middle ground between MMSODA's 'scemua' and 'reset' modes: instead of placing all of the confidence on one or the other, a little confidence is placed on both the simulated and the observed value of the states. A weighted average of the two is then calculated, and the simulated state (i.e. the prior state) is adjusted in the direction of the observed state. The magnitude of the adjustment is sometimes referred to as the 'state *innovation*' or simply 'state *update*'. Adding the state update to the prior state results in a new state value, which is known as the '*posterior* state' or the '*analysis* state'.

In terms of setup, the 'soda' mode is not so different from the 'reset' mode that you are familiar with. In fact, a quick look at the table of configuration variables in `mmsoda`'s documentation reveals that only 3 additional configuration variables are needed: `covObsPert`, `covModelPert`, and `nMembers`.

► 94. Read the description in `mmsoda`'s documentation for these three options.

► 95. Include `covObsPert`, `covModelPert`, and `nMembers` in your 'makeconf.m', and assign them values of `0.02`, `0.10` and `10`, respectively. Don't forget to change `modeStr` to `'soda'`, otherwise MMSODA will overwrite the values of `covObsPert` and `covModelPert` with `0`. Set `nModelEvalsMax` to `150` for now, re-run 'makeconf.m', and verify that './results/conf.mat' was indeed written.

► 96. Re-read the documentation on the objective function (see Figure 4.6), in particular the description of the input argument `modelOutput`, which is 3-D if `modeStr` is `'soda'`.

► 97. Adapt the objective function such that the SSR is calculated based on the ensemble-member mean of the prior state. Note: calculating the mean over the 3rd dimension of an N-dimensional array `X` in MATLAB is easily achieved by `mu = mean(X,3)`.

► 98. Test your configuration locally until it works. Then set `nModelEvals` to `7000` and re-run 'makeconf.m'.

► 99. Call `mmsodaPrepParallelFiles` to write the jobscript and the Makefile.

► 100. Copy your 'example4' working directory to the cluster.

4.7 MMSODA in 'soda' mode; parallel execution

► 101. Use PuTTY to set up an SSH connection to the LISA cluster. Make sure the necessary modules are loaded before compiling the binary. Submit your jobscript to the queue and wait for the optimization to finish.

► 102. Once the job finishes, edit 'makeconf.m' on your local machine by setting `saveEnKFResults` to `true` and `startFromUniform` to `false`. Increase the maximum number of model evaluations by `500`. Run `makeconf` and copy the new MMSODA configuration to the remote system. Submit the jobscript to the queue once more, and wait for the results.

► 103. Copy the optimization results back to your local machine.

► 104. Copy the code snippet from Listing 4.7 to your working directory. Adapt the filename from which the optimization results are read, and run the snippet on the results of the 'soda' mode optimization. The result should look like Fig. 4.10.

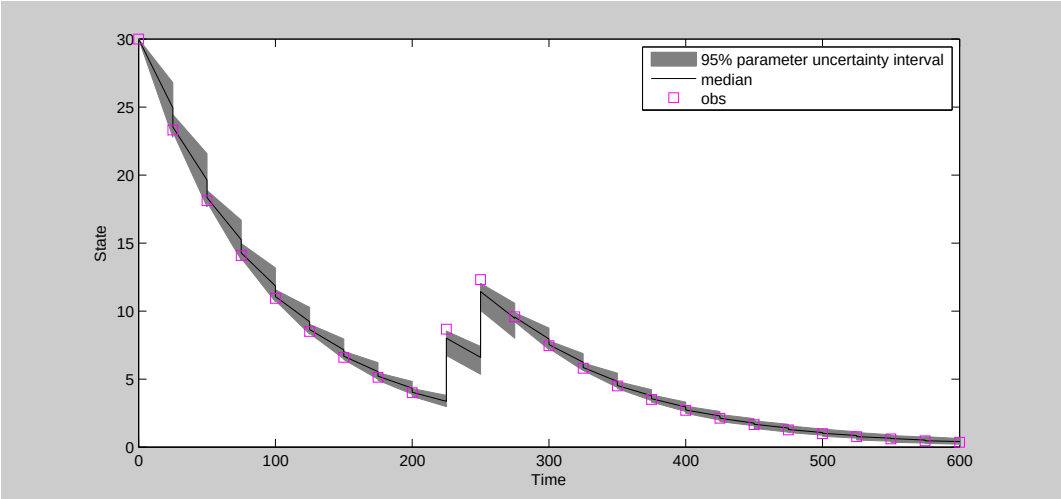


Figure 4.10: Parameter uncertainty intervals for the 'soda' mode optimization of the lintank model.

Appendices

Appendix A

Remote graphical applications

Up to this point, we've issued commands through the terminal. The terminal is a powerful tool, but sometimes it's also useful to run graphical programs (as opposed to text-based terminal programs) remotely, for example when you want to use the graphical debugging capabilities that the MATLAB GUI offers to debug some of your code on LISA, or if you want to view remotely generated MATLAB figures. Amazingly, this is possible through a process called *X forwarding over SSH*. Here, 'X' refers to the program that takes care of visualization on Linux. The Linux operating system is telling X what to visualize, and then X figures out which pixels on what screen should be what color, and whether a particular pixel is part of, say, a button, drop-down list, or some other user interface element. It is possible to re-route messages to the X system through your SSH connection, such that they end up on your local machine. If your local system is Linux, the X messages that your system receives can be interpreted by your local copy of the X program, and the user interface of a graphical application that is running on the remote machine will be displayed locally. However, you are probably running Windows, so you need to install a Windows version of X first.

Download Xming from http://sourceforge.net/projects/xming/files/Xming/6.9.0.31/Xming-6-9-0-31-setup.exe/download and install it. Now go to the Windows start menu and click XLaunch (Fig. A.1). This will bring up a wizard (Fig. A.2). Follow the steps outlined in Figs. A.3–A.5.

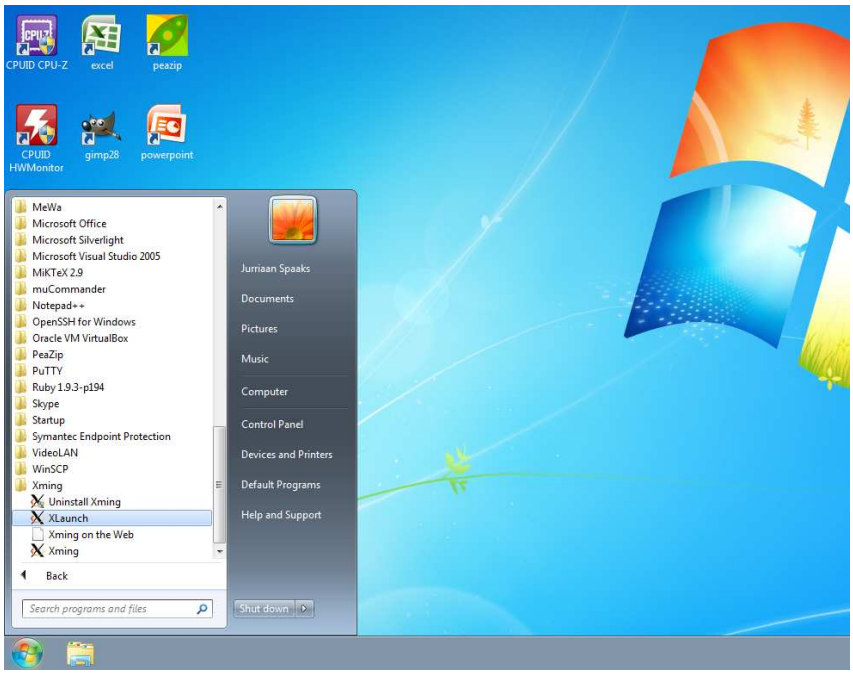


Figure A.1: Starting XLaunch from the Windows 7 start menu.

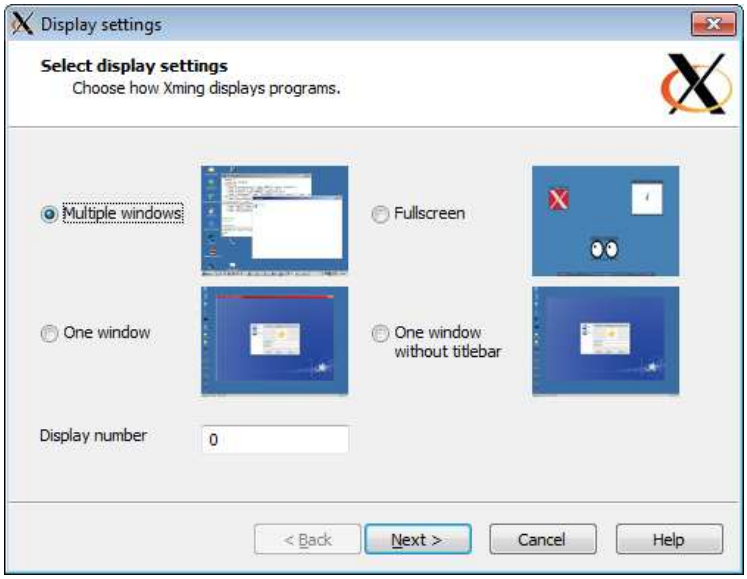


Figure A.2: The XLaunch configuration wizard page 1.

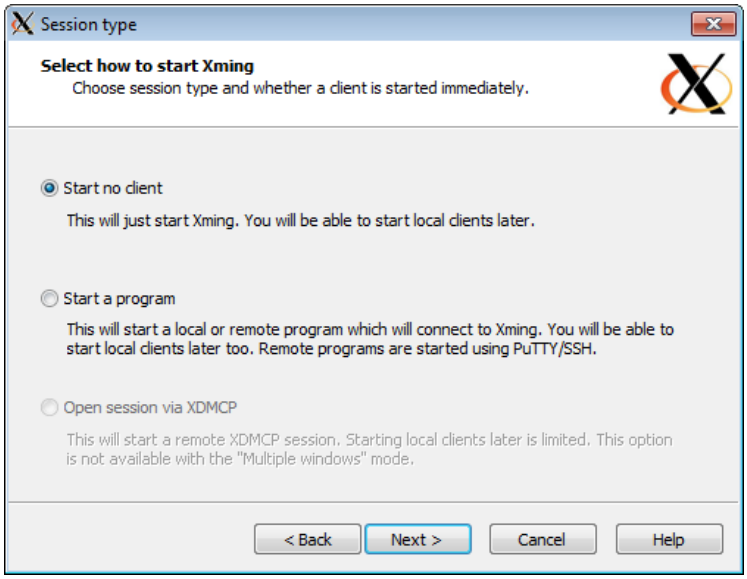


Figure A.3: The XLaunch configuration wizard page 2.

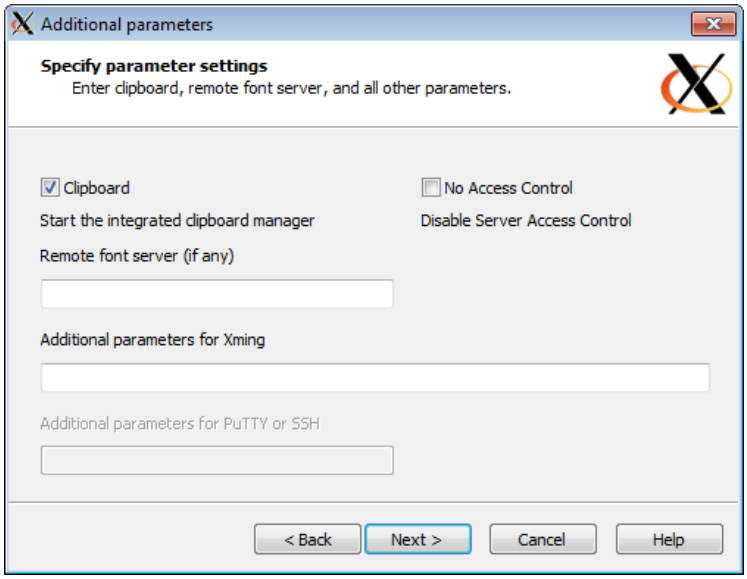


Figure A.4: The XLaunch configuration wizard page 3.

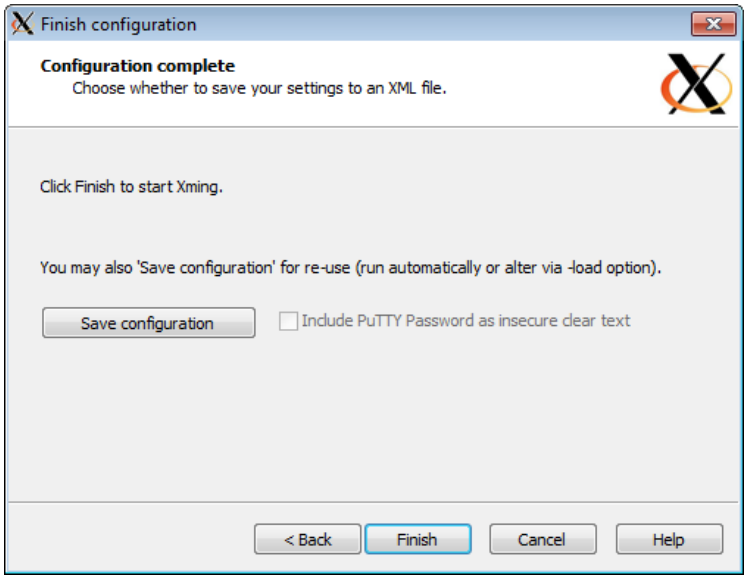


Figure A.5: The XLaunch configuration wizard page 4.

After you go through the XLaunch wizard, there should be an X icon in your icon tray.

Now that we have an X server for Windows running locally, we still need to tell the remote system that we want it to route its X messages through the SSH connection to our local system. For this, you need to start a new SSH connection. Start PuTTY, and type in the 'lisa.surfsara.nl' host name, exactly as before (recall Fig. 2.4). However, before clicking the 'Open' button, expand the plus sign symbol for 'SSH' in the bottom part of the left pane (see Fig. A.6). Find the item labeled 'X11' (X is sometimes referred to as 'X11'), and in the right pane, enable the checkbox that says 'Enable X11 forwarding' before clicking the 'Open' button.

At the prompt, you can quickly test whether the remote system is connected to your local X server by typing:

```
jspaaks@login2:~$ echo $DISPLAY
```

which should result in a message like this (the numbers could be different):

```
localhost:10.0
```

If the connection was unsuccessful, the shell returns an empty message.

Start Octave in silent mode by:

```
jspaaks@login2:~$ octave --silent
octave:1>
```

and then run the `peaks(30)` function.

```
octave:1> peaks(30)
octave:2>
```

After a few moments (depending on the bandwidth of your connection to LISA), it will show you Fig. A.7.

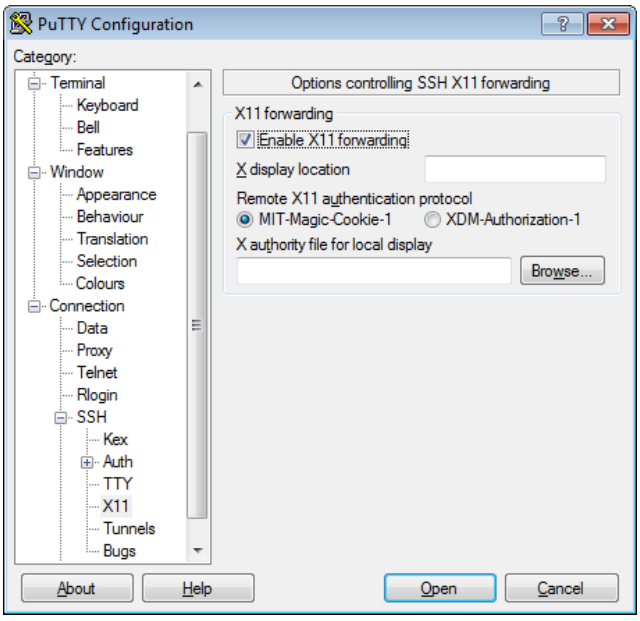


Figure A.6: Configure PuTTY for X forwarding over SSH.

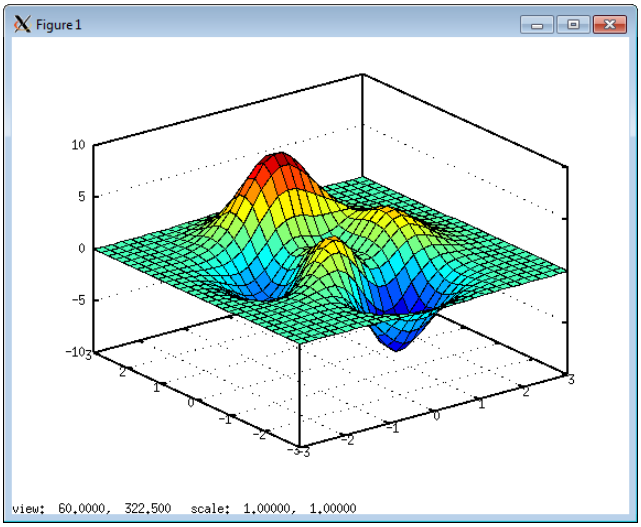


Figure A.7: After Octave generated this `peaks(30)` figure remotely, the remote system sent it to the local machine over SSH where it was displayed with Xming.

Let's see if we can do the same thing in MATLAB:

```
octave:2> exit

jspaaks@login2:~$ matlab
-bash: matlab: command not found
```

The reason this does not work is that MATLAB is only used by some users, therefore it is

not available by default[1]. However, it's easy enough to make MATLAB available, like so:

```
jspaaks@login2:~$ module load matlab
jspaaks@login2:~$ matlab
```

It is considered good practice to `unload` MATLAB once you are done with it:

```
jspaaks@login2:~$ module unload matlab
```

This frees up one of the MATLAB licenses, such that other users may use it. There are currently enough MATLAB licenses to run 32 instances of MATLAB simultaneously; however, some of the toolboxes require a separate license. For some toolboxes there are only 3 licenses available, so if your program happens to use one of those, you can only use 3 MATLAB instances simultaneously.

If you are just using the MATLAB licenses to run your programs on LISA, as opposed to doing development work, you can (legally) circumvent licensing problems by compiling your m-code, and running the compiled code with the *MATLAB Compiler Runtime* or *MCR*. The MCR can be loaded in a similar fashion as the full MATLAB suite:

```
jspaaks@login2:~$ module load mcr
```

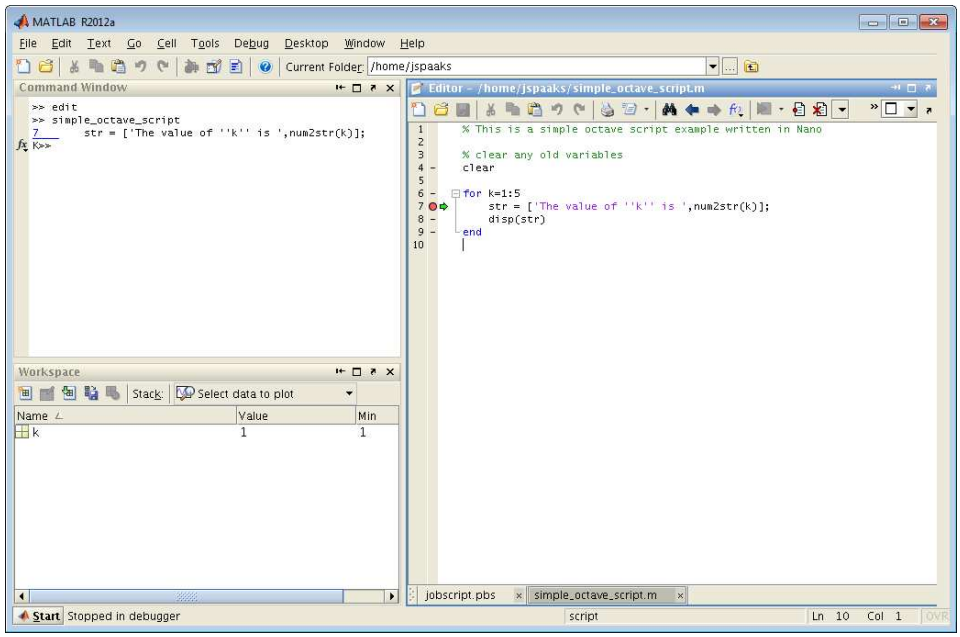


Figure A.8: Even MATLAB's debugging capabilities can be used remotely!

[1] In fact, you must be a member of the 'MATLAB users group' that exist on LISA. Any of LISA's administrators can add you to this group, but it usually involves some administration.

Appendix B

Likelihoods in optimization

B.1 Definition of the system

The discrete-time state-space formulation of a nonlinear dynamic system is[1]:

$$x_{t+1} = f(x_t, u_t, \boldsymbol{\theta}) \tag{B.1}$$

$$y_{t+1} = h(x_{t+1}, \boldsymbol{\phi}) \tag{B.2}$$

To keep notation as simple as possible, we consider a system in which the state, the forcing and the output at any given time can be represented as scalars. Further note that the 'state' of the system at a given time can include history. Since it is generally impossible to observe the true state, the true forcing, and the true output of a system, we have to make do with the observed state, observed forcing, and the observed output instead:

$$\tilde{x}_{t+1} = x_{t+1} + \omega_{t+1} \tag{B.3}$$

$$\tilde{u}_{t+1} = u_{t+1} + \psi_{t+1} \tag{B.4}$$

$$\tilde{y}_{t+1} = y_{t+1} + \nu_{t+1} \tag{B.5}$$

Note that Eqs. B.3–B.5 assume that the dimensionality of x_{t+1}, u_{t+1} and y_{t+1} is the same as their observed counterparts, and that they do indeed represent the same entities (in other words, there is no *incommensurability*). Furthermore, we do not know the mechanism by which x_t leads to x_{t+1} in Eq. B.1, so instead we propose a mechanism $\hat{f}(\cdot)$; similarly we do not know how x_{t+1} leads to y_{t+1} in Eq. B.2, so we propose $\hat{h}(\cdot)$. Typically, $\hat{f}(\cdot)$ and $\hat{h}(\cdot)$ are part of the same computer model structure. Because of philosophical reasons (e.g. Popper, 2009), it is impossible to prove that $\hat{f}(\cdot)$ and $\hat{h}(\cdot)$ are in fact the correct functions $f(\cdot)$ and

[1] Refer to Appendix C for the meaning of the symbols used in this Chapter.

$h(\cdot)$—instead, we can only subject $\hat{f}(\cdot)$ and $\hat{h}(\cdot)$ to increasingly difficult tests, and if $\hat{f}(\cdot)$ and $\hat{h}(\cdot)$ are not falsified by these tests, then the confidence in the correctness of $\hat{f}(\cdot)$ and $\hat{h}(\cdot)$ increases.

B.2 Constructing a likelihood for a simple error model

The probability of sampling a value x from a normal distribution is calculated with:

$$p\,(\,x\,|\,\mu,\sigma\,) = \frac{1}{\sqrt{2\pi\sigma^2}}\,e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{B.6}$$

Within the context of calibration, this is equivalent to:

$$p\,(\,\hat{x}\,|\,\tilde{x},\sigma\,) = \frac{1}{\sqrt{2\pi\sigma^2}}\,e^{-\frac{1}{2}\left(\frac{\hat{x}-\tilde{x}}{\sigma}\right)^2} \tag{B.7}$$

Note that this assumes that the true mean of the distribution can be observed, i.e. $\tilde{x} = \mu$.

For the case where we have not just 1 observation, but instead have a time series of n_o observations, the probability $p\,(\,\hat{\mathbf{x}}\,|\,\tilde{\mathbf{x}},\boldsymbol{\sigma}\,)$ is calculated as the product of all individual probabilities[1]:

$$p\,(\,\hat{\mathbf{x}}\,|\,\tilde{\mathbf{x}},\sigma\,) = \prod_{t=1}^{n_o} \frac{1}{\sqrt{2\pi\sigma^2}}\,e^{-\frac{1}{2}\left(\frac{\hat{x}_t-\tilde{x}_t}{\sigma}\right)^2} \tag{B.8}$$

Note that the multiplication of individual probabilities in Eq. B.7 reflects the implicit assumption that the errors are independent—an assumption that is often violated.

Since $\frac{1}{\sqrt{2\pi\sigma^2}}$ is constant for homoscedastic problems, Eq. B.7 can be rearranged as follows:

$$p\,(\,\hat{\mathbf{x}}\,|\,\tilde{\mathbf{x}},\sigma\,) = \left[\frac{1}{\sqrt{2\pi\sigma^2}}\right]^{n_o} \cdot \prod_{t=1}^{n_o} e^{-\frac{1}{2}\left(\frac{\hat{x}_t-\tilde{x}_t}{\sigma}\right)^2} \tag{B.9}$$

and since:

$$\left[\frac{1}{\sqrt{2\pi\sigma^2}}\right]^{n_o} = \left[\sqrt{2\pi\sigma^2}\right]^{-n_o}$$

[1] Note that this can be extended to deal with non-constant variance (*heteroscedasticity*) by making σ into a vector $\boldsymbol{\sigma}$:

$$p\,(\,\hat{\mathbf{x}}\,|\,\tilde{\mathbf{x}},\boldsymbol{\sigma}\,) = \prod_{t=1}^{n_o} \frac{1}{\sqrt{2\pi\sigma_t^2}}\,e^{-\frac{1}{2}\left(\frac{\hat{x}_t-\tilde{x}_t}{\sigma_t}\right)^2}$$

Eq. B.9 can be rewritten as:

$$p\left(\hat{\mathbf{x}}\,|\,\tilde{\mathbf{x}},\sigma\right) = \left[\sqrt{2\pi\sigma^2}\right]^{-n_o} \cdot \prod_{t=1}^{n_o} e^{-\frac{1}{2}\left(\frac{\hat{x}_t-\tilde{x}_t}{\sigma}\right)^2} \tag{B.10}$$

Furthermore,

$$e^a \cdot e^b = e^{a+b} \tag{B.11}$$

so Eq. B.10 may be written as:

$$p\left(\hat{\mathbf{x}}\,|\,\tilde{\mathbf{x}},\sigma\right) = \left[\sqrt{2\pi\sigma^2}\right]^{-n_o} \cdot e^{-\frac{1}{2}\sum_{t=1}^{n_o}\left(\frac{\hat{x}_t-\tilde{x}_t}{\sigma}\right)^2} \tag{B.12}$$

The probability density in Eq. B.12 is related to the *log-likelihood* according to[1]:

$$\ell\left(\hat{\mathbf{x}}\,|\,\tilde{\mathbf{x}},\sigma\right) = \ln\left(p\left(\hat{\mathbf{x}}\,|\,\tilde{\mathbf{x}},\sigma\right)\right) \tag{B.13}$$

$$= \ln\left(\left[\sqrt{2\pi\sigma^2}\right]^{-n_o} \cdot e^{-\frac{1}{2}\sum_{t=1}^{n_o}\left(\frac{\hat{x}_t-\tilde{x}_t}{\sigma}\right)^2}\right) \tag{B.14}$$

and since:

$$\ln\left(a\cdot b\right) = \ln\left(a\right) + \ln\left(b\right), \tag{B.15}$$

$$\ln\left(a^b\right) = b\cdot\ln\left(a\right), \tag{B.16}$$

$$\ln\left(e^a\right) = a, \tag{B.17}$$

Eq. B.13 can be rewritten as:

$$\ell\left(\hat{\mathbf{x}}\,|\,\tilde{\mathbf{x}},\sigma\right) = -n_o\cdot\ln\left[\sqrt{2\pi\sigma^2}\right] + \left[-\frac{1}{2}\sum_{t=1}^{n_o}\left(\frac{\hat{x}_t-\tilde{x}_t}{\sigma}\right)^2\right] \tag{B.18}$$

Subsequently applying Eqs. B.16 and B.15, Eq. B.18 can be rewritten as:

$$\ell\left(\hat{\mathbf{x}}\,|\,\tilde{\mathbf{x}},\sigma\right) = -\frac{1}{2}n_o\cdot\ln\left(2\pi\right) - \frac{1}{2}n_o\cdot\ln\left(\sigma^2\right) - \frac{1}{2}\sum_{t=1}^{n_o}\left(\frac{\hat{x}_t-\tilde{x}_t}{\sigma}\right)^2 \tag{B.19}$$

to yield the Gaussian log-likelihood function with unknown standard deviation σ of the residuals $\hat{x}_t - \tilde{x}_y$. It is convenient to rewrite Eq. B.19 as follows:

$$\ell\left(\hat{\mathbf{x}}\,|\,\tilde{\mathbf{x}},\sigma\right) = -\frac{1}{2}n_o\cdot\ln\left(2\pi\right) - \frac{1}{2}n_o\cdot\ln\left(\sigma^2\right) - \frac{1}{2}\sum_{t=1}^{n_o}\frac{\left(\hat{x}_t-\tilde{x}_t\right)^2}{\sigma^2} \tag{B.20}$$

For many measuring devices, σ^2 is either known or can be determined by simple experiments.

[1] Note that $\hat{\mathbf{x}}$ in Eq. B.13 is only dependent on the parameter vector $\boldsymbol{\theta}$, while the observations $\tilde{\mathbf{x}}$ and σ are given. When a probability is a function of the parameter value, 'likelihood' is preferred over 'probability'.

If necessary, σ^2 can also be estimated from the observations according to[1]:

$$s^2 = \frac{1}{n_o - 1} \sum_{t=1}^{n_o} \left(\hat{x}_t - \tilde{x}_t\right)^2 \tag{B.21}$$

Replacing σ^2 with s^2 in Eq. B.22 yields:

$$\ell\left(\hat{\mathbf{x}} \,|\, \tilde{\mathbf{x}}\right) = -\frac{1}{2} n_o \cdot \ln\left(2\pi\right) \; - \; \frac{1}{2} n_o \cdot \ln\left(s^2\right) \; - \; \frac{1}{2} \sum_{t=1}^{n_o} \frac{\left(\hat{x}_t - \tilde{x}_t\right)^2}{s^2} \tag{B.22}$$

which is equal to:

$$\ell\left(\hat{\mathbf{x}} \,|\, \tilde{\mathbf{x}}\right) = -\frac{1}{2} n_o \cdot \ln\left(2\pi\right) - \frac{1}{2} n_o \cdot \ln\left(\frac{1}{n_o - 1} \sum_{t=1}^{n_o} \left(\hat{x}_t - \tilde{x}_t\right)^2\right) - \frac{1}{2} \frac{\sum_{t=1}^{n_o} \left(\hat{x}_t - \tilde{x}_t\right)^2}{\frac{1}{n_o - 1} \sum_{t=1}^{n_o} \left(\hat{x}_t - \tilde{x}_t\right)^2} \tag{B.23}$$

Simplification of the last term in Eq. B.23 yields:

$$\ell\left(\hat{\mathbf{x}} \,|\, \tilde{\mathbf{x}}\right) = -\frac{1}{2} n_o \cdot \ln\left(2\pi\right) \; - \; \frac{1}{2} n_o \cdot \ln\left(\frac{1}{n_o - 1} \sum_{t=1}^{n_o} \left(\hat{x}_t - \tilde{x}_t\right)^2\right) \; - \; \frac{1}{2} \left(n_o - 1\right) \tag{B.24}$$

Application of Eq. B.15 to the second term in Eq. B.24 yields:

$$-\frac{1}{2} n_o \cdot \ln\left(\frac{1}{n_o - 1} \sum_{t=1}^{n_o} \left(\hat{x}_t - \tilde{x}_t\right)^2\right) = -\frac{1}{2} n_o \cdot \ln\left(\frac{1}{n_o - 1}\right) - \frac{1}{2} n_o \cdot \ln\left(\sum_{t=1}^{n_o} \left(\hat{x}_t - \tilde{x}_t\right)^2\right) \tag{B.25}$$

and since

$$\ln\left(\frac{a}{b}\right) = \ln\left(a\right) \; - \; \ln\left(b\right) \tag{B.26}$$

and

$$\ln(1) = 0, \tag{B.27}$$

Eq. B.25 thus becomes:

$$-\frac{1}{2} n_o \cdot \ln\left(\frac{1}{n_o - 1} \sum_{t=1}^{n_o} \left(\hat{x}_t - \tilde{x}_t\right)^2\right) = +\frac{1}{2} n_o \cdot \ln\left(n_o - 1\right) - \frac{1}{2} n_o \cdot \ln\left(\sum_{t=1}^{n_o} \left(\hat{x}_t - \tilde{x}_t\right)^2\right) \tag{B.28}$$

Filling Eq. B.28 back into Eq. B.24 yields:

$$\ell\left(\hat{\mathbf{x}} \,|\, \tilde{\mathbf{x}}\right) = -\frac{1}{2} n_o \cdot \ln\left(2\pi\right) + \frac{1}{2} n_o \cdot \ln\left(n_o - 1\right) - \frac{1}{2} n_o \cdot \ln\left(\sum_{t=1}^{n_o} \left(\hat{x}_t - \tilde{x}_t\right)^2\right) - \frac{1}{2} \left(n_o - 1\right) \tag{B.29}$$

[1] This implicitly assumes that parameter uncertainty is the only source of uncertainty.

For a given number of observations n_o, terms 1, 2, and 4 from Eq. B.29 may be collected into a constant C as follows:

$$\ell\,(\,\hat{\mathbf{x}}\,|\,\tilde{\mathbf{x}}\,) = -\,\frac{1}{2}n_o \cdot \ln\left(\sum_{t=1}^{n_o}(\hat{x}_t - \tilde{x}_t)^2\right) \;+\; C \tag{B.30}$$

with

$$C = -\frac{1}{2}n_o \cdot \ln(2\pi) \;+\; \frac{1}{2}n_o \cdot \ln(n_o - 1) \;-\; \frac{1}{2}(n_o - 1) \tag{B.31}$$

When constructing log likelihood functions for MMSODA, we may safely leave out the C term. This is because MMSODA uses the Metropolis algorithm to determine whether a new sample of the parameter space should be accepted or rejected. Whether a new sample is accepted depends on how its likelihood ℓ_{new} compares to the likelihood associated with a previous sample ℓ_{prev}. A sample is accepted if

$$\ell_{new} - \ell_{prev} \;>\; \ln(z) \tag{B.32}$$

with z a draw from a uniform distribution between 0 and 1:

$$z \;\sim\; U(0,1) \tag{B.33}$$

In other words, if ℓ_{new} is an improvement relative to ℓ_{prev}, the new point is always accepted; if ℓ_{new} is slightly worse than ℓ_{prev}, it is still quite likely that the new point will be accepted; if ℓ_{new} is much worse than ℓ_{prev} it is unlikely, but not impossible, that the new point will be accepted.

Since raising both ℓ_{new} and ℓ_{prev} by the constant C has no effect on the distance $\ell_{new} - \ell_{prev}$, C may be left out of the objective function entirely.

Appendix C

Definition of terms

Table C.1: Definition of terms.

e^x	Euler's e raised to the power x, or in MATLAB syntax `exp(x)`
$\ln(x)$	natural logarithm of a variable x, or in MATLAB syntax `log(x)`
$f(\cdot)$	true state operator (nonlinear function)
$\hat{f}(\cdot)$	supposed state operator (nonlinear function)
$h(\cdot)$	true measurement operator (nonlinear function)
$\hat{h}(\cdot)$	supposed measurement operator (nonlinear function)
$\boldsymbol{\theta}$	true parameter values pertaining to the state operator
$\hat{\boldsymbol{\theta}}$	estimate of the parameter values pertaining to the state operator
$\boldsymbol{\phi}$	true parameter values pertaining to the measurement operator
σ	standard deviation of the normal distribution
μ	mean of the normal distribution
ν_{t+1}	random perturbation of the true output at time t
ψ_{t+1}	random perturbation of the true forcing at time t
ω_{t+1}	random perturbation of the true state at time t
u_t	true forcing at time t
$\tilde{u}_t$	observed forcing at time t
x_t	true state at time t
$\tilde{x}_t$	observed state at time t

Continued on next page

Table C.1: Definition of terms (continued).

$\hat{x}_t$	predicted state at time t
y_t	true output at time t
$\tilde{y}_t$	observed output at time t
$\hat{y}_t$	predicted output at time t
$\boldsymbol{\varepsilon}$	observation-prediction residual vector
$\mathcal{N}(\mu, \sigma^2)$	normal probability distribution with mean μ and variance σ^2
$\ell(\cdot)$	log likelihood
s^2	estimator of variance σ^2
n_o	number of observations in the vector $\tilde{\mathbf{x}}$

Bibliography

Martyn P. Clark and Jasper A. Vrugt. Unraveling uncertainties in hydrologic model calibration: Addressing the problem of compensatory parameters. *Geophysical Research Letters*, 33:L06406, 2006. doi: 10.1029/2005GL025604.

Karl Popper. *The Logic of Scientific Discovery*. Routledge, 2009. first published as *Logik der Forschung*, 1935 by Verlag von Julius Springer, Vienna, Austria.

Rainer Storn and Kenneth Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11:341–359, 1997.

J. A. Vrugt, H. V. Gupta, L. A. Bastidas, W. Bouten, and S. Sorooshian. Effective and efficient algorithm for multi-objective optimization of hydrologic models. *Water Resources Research*, 39(8):1214, 2003a. doi: 10.1029/2002WR001746.

J. A. Vrugt, H. V. Gupta, W. Bouten, and S. Sorooshian. A Shuffled Complex Evolution Metropolis algorithm for optimization and uncertainty assessment of hydrologic model parameters. *Water Resources Research*, 39(8):1201, 2003b. doi: 10.1029/2002WR001642.

Jasper A. Vrugt, Cees G. H. Diks, Hoshin V. Gupta, Willem Bouten, and Jacobus M. Verstraten. Improved treatment of uncertainty in hydrologic modeling: Combining the strengths of global optimization and data assimilation. *Water Resources Research*, 41: W01017, 2005a. doi: 10.1029/2004WR003059.

Jasper A. Vrugt, Bruce A. Robinson, and Velimir V. Vesselinov. Improved inverse modeling for flow and transport in subsurface media: Combined parameter and state estimation. *Geophysical Research Letters*, 32:L18408, 2005b. doi: 10.1029/2005GL023940.

Jasper A. Vrugt, Hoshin V. Gupta, Breandánn Ó Nualláin, and Willem Bouten. Real-time data assimilation for operational ensemble streamflow forecasting. *Journal of Hydrometeorology*, 7(3):548–575, 2006a. doi: 10.1175/JHM504.1.

Jasper A. Vrugt, Breandánn Ó Nualláin, Bruce A. Robinson, Willem Bouten, Stefan C. Dekker, and Peter M. A. Sloot. Application of parallel computing to stochastic parameter estimation in environmental models. *Computers & Geosciences*, 32:1139–1155, 2006b. doi: 10.1016/j.cageo.2005.10.015.

Jasper A. Vrugt, Philip H. Stauffer, Th. Wöhling, Bruce A. Robinson, and Velimir V. Vesselinov. Inverse modeling of subsurface flow and transport properties: A review with new developments. *Vadose Zone Journal*, 7(2):843–864, 2008. doi: 10.2136/vzj2007.0078.

Jasper A. Vrugt, C. J. F. Ter Braak, C. G. H. Diks, Bruce A. Robinson, James M. Hyman, and Dave Higdon. Accelerating Markov Chain Monte Carlo simulation by Differential Evolution with self-adaptive randomized subspace sampling. *International Journal of Nonlinear Sciences and Numerical Simulation*, 10(3):271–288, 2009.

Index