UNIVERSITY OF AMSTERDAM

esibayes manual

Author: Jurriaan H. Spaaks

Contents

Chapter 1

Introduction

This document is a manual for learning how to use the LISA cluster computer hosted at SARA (http://www.sara.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 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 or Octave. 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.

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.sara.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.

Download WinSCP from http://sourceforge.net/projects/winscp/files/WinSCP/4.3.7/winscp437.zip/download[2,3]. Unzip into your 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.sara.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.sara.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).

Download 'tutorial.zip' from http:\www.????.?? and unzip into your (local) home directory or a USB drive.

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.

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

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

[2] 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/.

[3] 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.sara.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.

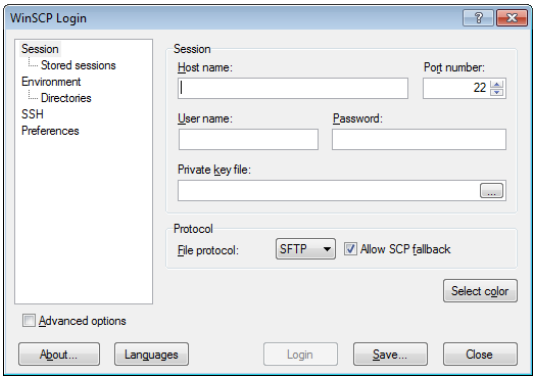

Figure 2.2: WinSCP session dialog box.

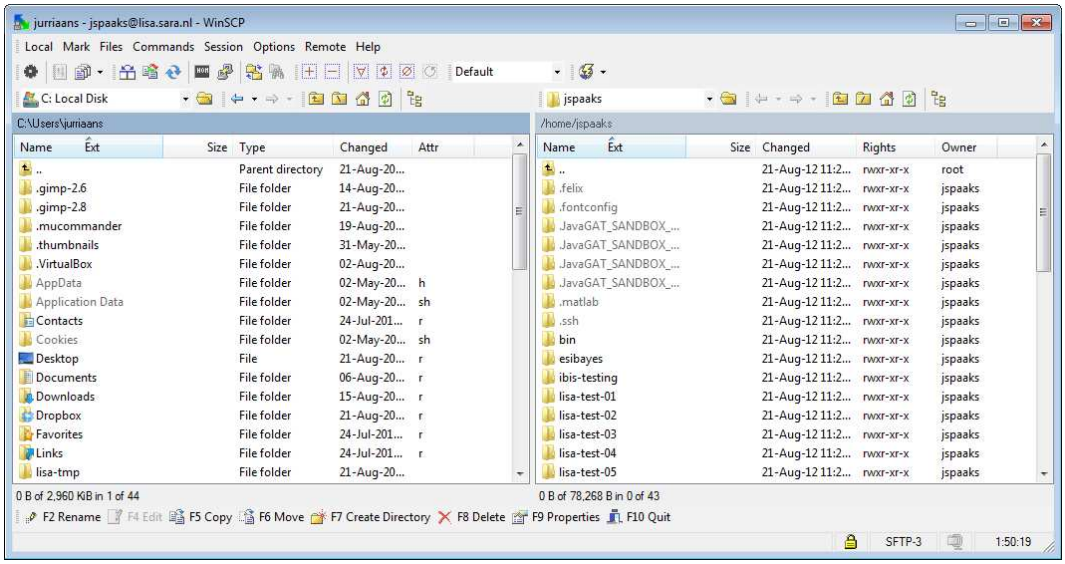

Figure 2.3: WinSCP interface. On the left is my local file system, on the right is my home directory on the remote file system. If this is the first time you connect to LISA, the remote file system should be pretty much empty.

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 it 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.sara.nl' and press the 'Open' button. PuTTY first prompts you for your username and then for your password. Fill in your LISA credentials.[2]

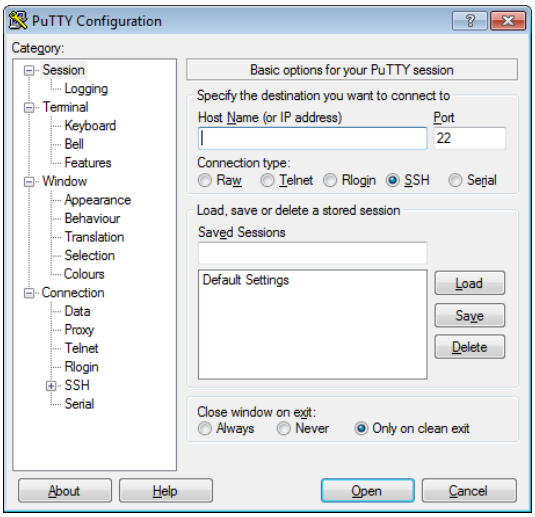

Figure 2.4: PuTTY session dialog box.

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

```
jspaaks@login4:~$
```

where `jspaaks` is the username and `login4` is the name of the remote machine. There are, in fact, two machines on which you can log in remotely: besides `login4`, there's also `login3`. For our intents and purposes, it doesn't matter whether you are logged in to `login3` or `login4`, but if you want to change from one to the other, you can do so with:

```
jspaaks@login4:~$ ssh login3
```

[1] See http://i.top500.org/stats 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.sara.nl`, substituing your own username in stead of `jspaaks`. Fill in your credentials when prompted.

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

```
jspaaks@login4:~$ 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@login4:~$ 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.

So, `pwd` 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:\' [1]. The first forward slash in the address is referred to as the 'file system root'. 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@login4:~$ ls
```

This should show at least the 'tutorial' folder that we just copied using WinSCP.

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

```
jspaaks@login4:~$ cd tutorial
jspaaks@login4:~/tutorial$
```

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

```
jspaaks@login4:~/tutorial/a/really/deeply/nested/directory$ cd ~
jspaaks@login4:~$
```

or if you want to move just one directory up:

```
jspaaks@login4:~/tutorial/a/really/deeply/nested/directory$ cd ..
jspaaks@login4:~/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:

[1] Somewhat confusingly, the account that has Administrator privileges is also referred to as 'root' on Linux systems.

```
jspaaks@login4:~$ mkdir anewdir
```

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

```
jspaaks@login4:~$ mkdir aNewDir
```

will result in an additional directory:

```
jspaaks@login4:~$ ls
anewdir   aNewDir   tutorial
jspaaks@login4:~$
```

The different approaches that Windows and Linux take in regard to 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:

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@login4:~$ rmdir aNewDir
```

or like this if you want to remove multiple directories:

```
jspaaks@login4:~$ rmdir anewdir aNewDir
```

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) 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@login4:~$ 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 `-`.

[1] You'll thank me later!

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@login4:~$ 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@login4:~$ 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@login4:~$ ls -l -h
```

or, combining the shorthand options:

```
jspaaks@login4:~$ ls -lh
```

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

```
jspaaks@login4:~$ 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@login4:~$ 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@login4:~$ 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@login4:~/tutorial/a/really/deeply/nested/directory$ cat with-a-file-in-it.txt
```

`cat` is short for 'concatentate', 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@login4:~$ cd tutorial/a/really/deeply/nested/directory/
jspaaks@login4:~/tutorial/a/really/deeply/nested/directory$ ls
with-a-file-in-it.txt
jspaaks@login4:~/tutorial/a/really/deeply/nested/directory$ cat with-a-file-in-it.txt
* * * * * * * * * * * * * * * * * * * * *
* *                                     * *
* *  hello world, the classic phrase    * *
* *                                     * *
* * * * * * * * * * * * * * * * * * * * *
```

```
jspaaks@login4:~/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@login4:~/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@login4:~/tutorial/a/really/deeply/nested/directory$ cd ~
jspaaks@login4:~$ 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@login4:~$
```

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@login4:~$ 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@login4:~$ mv with-a-file-in-it.txt the-renamed-file.txt
jspaaks@login4:~$ 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@login4:~$
```

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@login4:~$ 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@login4:~/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@login4:~/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@login4:~/tutorial/a$
```

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

```
1 jspaaks@login4:~$ cd tutorial
2 jspaaks@login4:~/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@login4:~/tutorial$ mkdir another
 7  jspaaks@login4:~/tutorial$ cp -R a/* another
 8  jspaaks@login4:~/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@login4:~/tutorial$ cd another/really/deeply/nested/directory/
14  jspaaks@login4:~/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 a Octave command window, like so:

```
jspaaks@login4:~$ 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'.

octave:1>
```

At the prompt, you can type any Octave command. 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:

```
jspaaks@login4:~$ 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.

```
  GNU nano 2.2.4                 New Buffer                                        Modified

% 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




^G Get Help    ^O WriteOut    ^R Read File   ^Y Prev Page   ^K Cut Text    ^C Cur Pos
^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@login4:~$ cat simple_octave_script.m
% 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@login4:~$
```

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

```
jspaaks@login4:~$ 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@login4:~$ 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@login4:~$
```

2.5 Jobscripts and the scheduler

So far, we've executed all our commands on just one machine, 'login4'. 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* or *batch file*. The jobscript lays out the requirements of the task, such as the number of nodes that is 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=00:01:00
2  #PBS -lnodes=1:cores8:ppn=1
3  #PBS -S /bin/bash
4
5  echo `date`: job starts
6
7  # run octave with the program:
8  octave --silent --no-window-system --eval "disp('hello world')"
9
10 echo `date`: job ends
11
12 exit
```

This jobscript asks the scheduler for a time slot of 1 minute (`lwalltime=00:01:00`), 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. As a matter of fact, you already know it, since the shell program you have been using is bash. The script then echoes the current date and time plus the message ': job starts' to the standard output (more about standard input and 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. 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@login4:~/tutorial/simple-batch$ qsub jobscript-slow.pbs
6388734.batch1.irc.sara.nl
jspaaks@login4:~/tutorial/simple-batch$
```

Note that I'm actually submitting a slightly different version of the jobscript here, which, as the name suggests, needs more time to finish. This is to make sure that the job runs sufficiently long for a user to check the job's statistics, because the LISA system can't show any statistics for jobs that are finished already.

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@login4:~/tutorial/simple-batch$ qstat 6388734
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
6388734.batch1            ...ript-slow.pbs jspaaks                 0 Q express
jspaaks@login4:~/tutorial/simple-batch$
```

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@login4:~/tutorial/simple-batch$ qstat -u jspaaks -n

batch1.irc.sara.nl:
                                                                  Req'd  Req'd   Elap
Job ID                  Username Queue    Jobname          SessID NDS   TSK Memory Time  S Time
----------------------- -------- -------- ---------------- ------ ----- --- ------ ----- - -----
6388734.batch1.i        jspaaks  express  jobscript-slow.p  15291     1   1    --  00:01 R 00:00
   gb-r2n14/0
jspaaks@login4:~/tutorial/simple-batch$
```

You can look up statistics for a given node at https://ganglia.sara.nl/?r=hour&cs=&ce=&c=LISA+Cluster&h=gb-r2n14.irc.sara.nl&tab=m&vn=&mc=2&z=small&metric_group=ALLGROUPS, 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 batch queue

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

3. the interactive queue

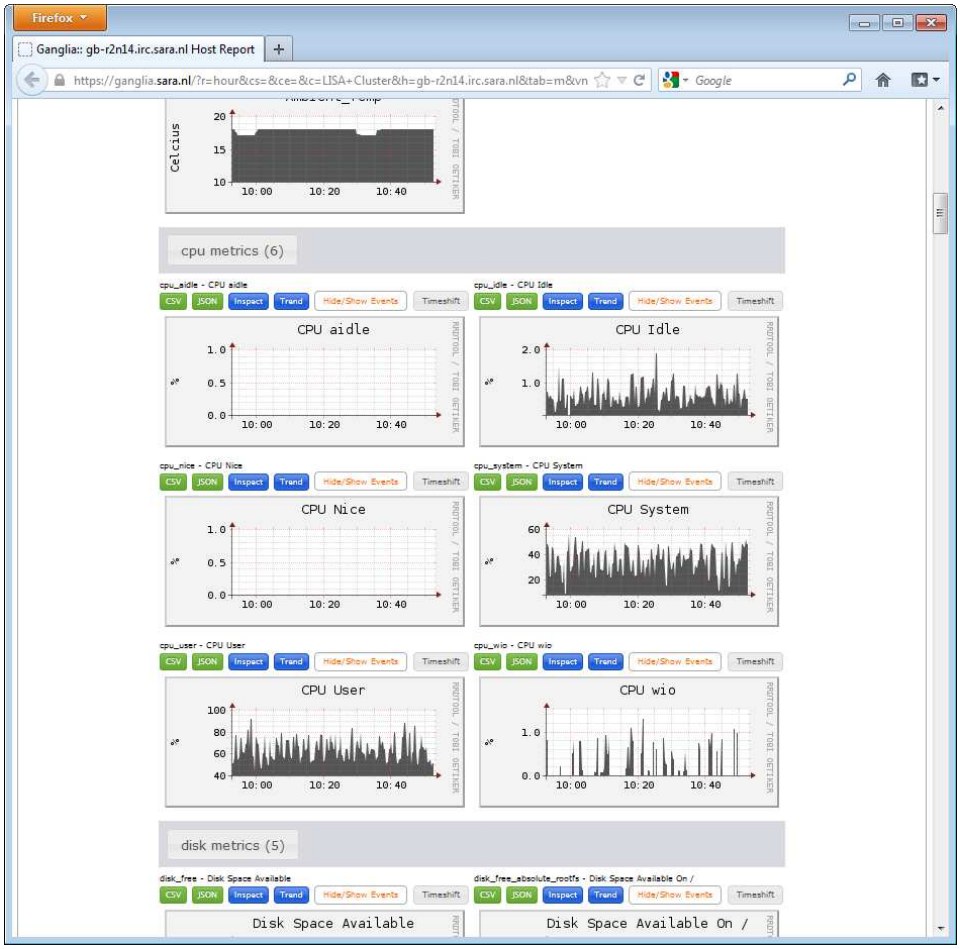

Figure 2.5: Ganglia visualizes a wide array of performance indices for the LISA system. The plots can be viewed in a web browser such as Firefox.

Your job ends up in the express queue if it asks for less than 15 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. In contrast, jobs that are in the batch queue can take a long time to start, but when they finally do, you can unleash the full computing power of the cluster. The interactive queue is primarily meant for development work that requires multiple machines. Submitting your job to the interactive queue is accomplished by the `-I` option:

```
jspaaks@login4:~/tutorial$ qsub -I -lnodes=2:cores8:ppn=1 -lwalltime=00:03:00
qsub: waiting for job 6388157.batch1.irc.sara.nl to start
```

The above command asks for an interactive session with 2 nodes of the core8 type, each of which will only be running 1 process. 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.sara.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 'login4' anymore, but on one of the computing nodes within the cluster. When your time is up, the following message will appear:

```
jspaaks@gb-r7n32:~$ echo $PBS_O=>> PBS: job killed: walltime 215 exceeded limit 180
```

and you will return to 'login4' (or 'login3' if that's where you were before).

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

```
jspaaks@login4:~/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@login4:~/tutorial/simple-batch$
```

Also, `showq` shows whether the cluster is very busy or not (Friday afternoons are usually the busiest, Mondays and Tuesdays are quiet in comparison). The statistics displayed by `qsub` are updated every 15 seconds or so.

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

```
jspaaks@login4:~/tutorial/simple-batch$ qdel 6388079
```

This deletes job `6388079` from the queue. Very useful if you accidentally requested too many nodes, or if your job crashed after like one second because you made a mistake. 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@login4:~/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@login4:~/tutorial/simple-batch$
```

Finding out when your job is scheduled to start goes like this (the estimate is quite inaccurate in my experience):

```
showstart 6388079
```

Chapter 3

Parallel optimization

Optimization algorithms such as SCEM-UA (Vrugt et al., 2003), SODA (Vrugt et al., 2005), 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, α is about 0.1 (not counting the overhead due to parallelization), therefore the theoretical maximum speedup is about 10x. For this example, a relatively large part of the time is spent in communication ('parallelization overhead'), the application is therefore *fine-grained*. If a parallel implementation of a program spends most of its time in calculation (as opposed to 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*.

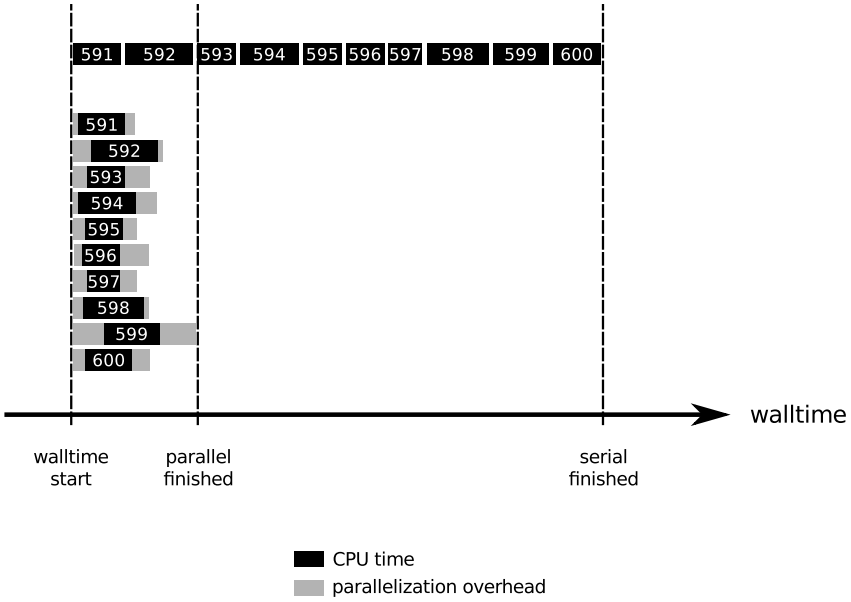

Figure 3.1: Walltime needed to complete 10 tasks 591–600 when evaluated in series or in parallel.

3.1 The Master-Worker paradigm

For Monte-Carlo type optimization, the Master-Worker paradigm (sometimes also referred to as Master-Slave) 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 initial and boundary conditions 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. Once it has received the parameter combination, it runs the model structure with the parameter combination it just received. 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.

Chapter 4

piped

In the previous chapter, I described the Master-Worker paradigm conceptually. In this chapter, we look at one specific piece of software that implements this concept. The software is called 'piped'. piped is a so-called *daemon*, i.e. a program that does not have an graphical interface but instead runs in the background[1]. Its task is to manage the communication between the master node and the workers (see Fig. 4.1). The Master now only has to tell piped which tasks it wants to evaluate, and wait for the results while the piped program takes care of the logistics of communicating with the Worker nodes.

You can check if `piped` is already running by checking the processes that are running system-wide:

```
jspaaks@login4:~$ ps -ef
```

and then filter the long list of results with the `grep` program:

```
jspaaks@login4:~$ ps -ef|grep ruby
jspaaks   17312 17193  0 00:14 pts/5    00:00:00 ruby1.9.1 ./piped.rb
jspaaks   17363 17193  0 00:14 pts/5    00:00:00 grep ruby
jspaaks@login4:~$
```

If there is an old `piped` still running, you must terminate it using the `kill` command followed by the identifier of the process, e.g.

```
jspaaks@login4:~$ kill 17312
```

otherwise the messages from different `piped` instances will get intermixed.

Besides piped, there are various other softwares that implement the Master-Worker paradigm. The most commonly used Master-Worker software is probably MPI, which stands for *Message Passing Interface*.

[1] http://en.wikipedia.org/wiki/Daemon_%28computing%29

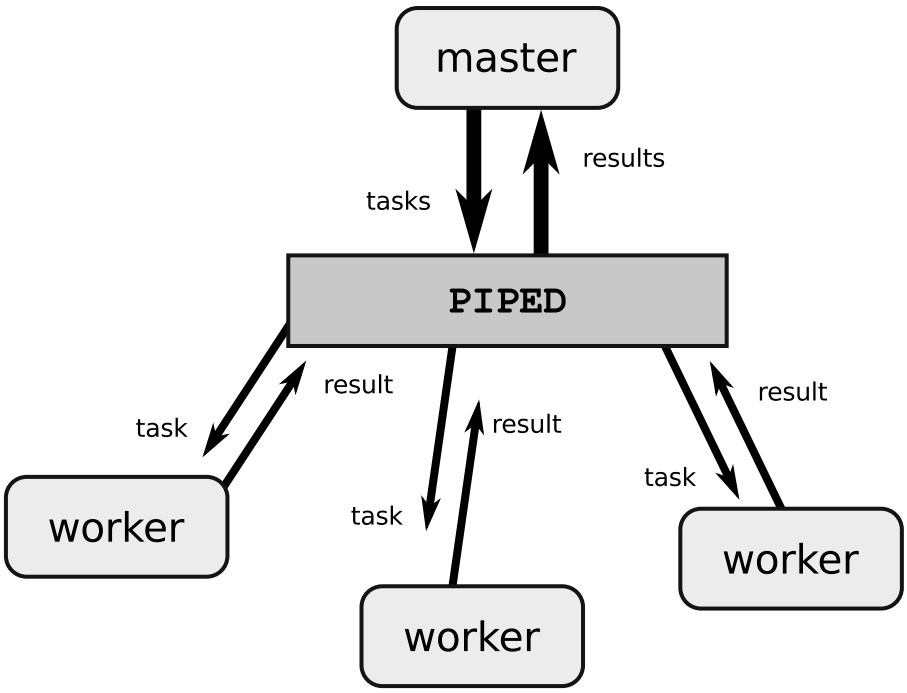

Figure 4.1: Message passing between a master node and its workers.

Chapter 5

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., 2005) 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 merges a number of previously separate softwares, namely SCEM-UA (Vrugt et al., 2003), MOSCEM-UA (Vrugt et al., 2003), SODA (Vrugt et al., 2005), MOSODA (), pSCEM-UA (), and pSODA (). In short, those acronyms imply 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 in series, or in parallel.

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. 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 the simplest possible example: a single-objective SCEM-UA optimization of a benchmark function `calcLikelihood()`, and let's not do anything in parallel just yet.

5.1 MMSODA in 'bypass' mode; offline

▶ 1. First download the `mmsoda` folder from <somewhere>.

▶ 2. Create a new directory next to your `mmsoda` folder. 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 folders, 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 the following subfolders:

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

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

▶ 4. 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 folder 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')`
on a Windows machine or
`>> addpath('/home/jspaaks/esibayes/mmsoda')`
on Linux.

▶ 5. At the MATLAB prompt, type:
`>> soda --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 MMSODA's main directory on your storage.

▶ 6. Test whether everything works as it should by typing:
`>> doc soda`
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.

▶ 7. Spend at least 12 minutes to browse through the documentation. In any case, make sure to read the documentation on 'soda.m'.

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

$$p = \frac{1}{2} \cdot \frac{1}{\sqrt{2 \cdot \pi \cdot \sigma_1}} \cdot \exp\left[-\frac{1}{2} \cdot \left(\frac{x-\mu_1}{\sigma_1}\right)^2\right] + \frac{1}{2} \cdot \frac{1}{\sqrt{2 \cdot \pi \cdot \sigma_2}} \cdot \exp\left[-\frac{1}{2} \cdot \left(\frac{x-\mu_2}{\sigma_2}\right)^2\right] \tag{5.1}$$

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

$$l = \log(p) \tag{5.2}$$

5.1.1 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 needed for running `soda()`.

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

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

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

▶ 10. At the prompt, type
```
doc soda
```
and bring up the HTML documentation for the `soda()` 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.

▶ 11. If you look in the table with all 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.

▶ 12. 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]`.

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

► 14. Save and close 'makeconf.m'.

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

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

► 16. 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 5.1–5.2.

► 17. 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.

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

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

Finally, we need to create the objective function m-file that implements equations 5.1 and 5.2.

5.1.2 Creating the objective function m-file

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

► 21. 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,allStateValuesKF,allValuesNOKF,parVec)`

► 22. As a second line, type:
`sodaUnpack()`
This function uses the information from the input arguments to construct the variable `x` and assigns it a value based on the value of `parVec`. Furthermore, it constructs the model constants and assigns 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;
```

▶ 23. Add this calculation to your `calcLikelihood` function.

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

▶ 25. Save and close 'calcLikelihood.m'.

5.1.3 Running the optimization locally

▶ 26. 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'.

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

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

▶ 29. Now, we are ready to run the optimization. At the MATLAB command prompt, type:
`>> [evalResults,critGelRub,sequences,metropolisRejects,conf] = soda();`
and wait for the optimization to finish. Input arguments to `soda` are not required, since `soda` 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.

▶ 30. *some exercises with interpretation of the results.*

▶ 31. Refer back to the documentation on the `soda` function; specifically, browse through some of the configuration options for running MMSODA in 'bypass' mode. Alter your 'makeconf.m' with the options that you find useful, and re-run the optimization.

▶ 32. 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 jobscript, which can be tricky to set up sometimes. Therefore, the MMSODA Toolbox for MATLAB comes with a function that helps you set up the jobscript correctly by asking a series of questions. At the MATLAB prompt, type:
`>> sodaWriteJobscript`
and use the following information to answer the questions:

1. the optimization will run on one of the login nodes;
2. we want not much verbal feedback from the program;
3. we want the default value for MPI_BUFFER_SIZE;
4. we don't want to use all the cores on the login node;
5. we want to start 8 processes;

6. we don't want to save the timing information.

(Note that the default answer is indicated in brackets, and that you can accept the default by simply pressing Enter.)

When `sodaWriteJobscript` 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.

5.2 MMSODA in 'bypass' mode; online

► 33. 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.

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 a script called 'Makefile' that we have prepared already. This script collects all the relevant software (your model files, your objective functions, as well as the code that enables communication between the Master and the Workers) and creates one program out of it.

5.3 Compiling MMSODA and your model code into a binary

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

► 35. In the PuTTY terminal, load the MATLAB program and MPI programs by typing:
`module load matlab`
and
`module load openmpi/gnu`
at the prompt. (These commands will not give any feedback on the success or otherwise of the command, but you could check that by typing the following command:
`module list`
which should now include both MATLAB and OpenMPI/GNU).

► 36. Use the `cd` command to set 'example1' as your current directory.

► 37. Now we are ready to compile. At the terminal, type:
`make`
You should see some text scrolling over your screen—it takes a while 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.

► 38. 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 'run-mmsoda.sh' file we just created on your local machine. 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, i.e. 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.

▶ 39. Change the permission bit for 'run-mmsoda.sh' by typing at the prompt:
`chmod +x run-mmsoda.sh`
Check that the permissions have changed to `-rwxr--r--` (`x` for 'execute').

▶ 40. Now we can start the optimization in parallel on the login node. At the terminal, type:
`./run-mmsoda.sh`
(Don't omit the `./` at the beginning, otherwise it won't work.)

▶ 41. *Parallel execution is actually slower than in series due to Amdahl's Law.*

5.4 MMSODA in 'scemua' mode; offline

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. The model we will use in this section is called HYMOD (e.g. **?**).

▶ 42. Go to tutorial/example2 and open hymod_batch.m in the MATLAB editor. Study the structure of the script.

Although 'hymod_batch' is a fairly simple model, it has all the ingredients typical of a dynamic model: it has an initial part, a dynamic part and a final part. In the initial part, the model parameters are specified. Furthermore, the state variables are assigned their initial value, and a number of other variables are created whose value does not change during the model run, but which are still necessary, i.e. model constants. Among these the `timeVec` variable is particularly important, since it defines the points in time for which a model prediction is required.

When you want to set up a dynamic model like HYMOD, such that it runs within an optimization framework like the MMSODA Toolbox for MATLAB, you need to make some changes to the model file(s). For starters, the script must be adapted such that the model prediction becomes a function of the model parameter vector, the constants, and the initial state values. During the conversion from model script to model function most of the initial part is usually moved elsewhere—we want the model function to directly start with the dynamic part.

▶ 43. First, let's create the 'constants.mat' file by adding a `save` command in 'hymod_batch.m'. (Check out `doc save` for information about saving only particular variables, i.e. only the variables that are in fact model constants). Running the 'hymod_batch' should create 'constants.mat' in tutorial/example2.

▶ 44. On your local machine, create a new directory next to your `mmsoda` and `example1` folders. Let's call this directory `example2`. Change directory into `example2`, and create the necessary subdirectories in it.

▶ 45. Copy your 'constants.m' into example2/data/

▶ 46. Set your MATLAB working directory to `example2`.

▶ 47. Take another look at the HTML documentation for `soda`, specifically at the table that lists the configuration variables for 'scemua' mode. The following variables are required for a 'scemua' optimization: `modeStr`, `modelName`, `objCallStr`, `parNames`, `parSpaceHiBound`, `parSpaceLoBound`, and `priorTimes`. Make sure you understand the description for each of these configuration variables.

▶ 48. 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. Use the information below to set it up correctly:

1. Set `modeStr` to `'scemua'`;
2. Set `modelName` to `'hymod'`;
3. Set `objCallStr` to `'calcLikelihoodState'`;
4. Set `parNames` to a cell array of strings with the 5 parameter names exactly as they are used in the dynamic part of the model;
5. For the upper boundary of the parameter space, use 350.0, 0.6, 0.99, 0.02, and 0.600 for `cmax`, `bexp`, `fQuickflow`, `Rs`, and `Rq`, respectively;
6. For the lower boundary of the parameter space, use 200.0, 0.1, 0.00, 0.001, and 0.200 for `cmax`, `bexp`, `fQuickflow`, `Rs`, and `Rq`, respectively;
7. Get the `priorTimes` values from `timeVec`.

5.4.1 Creating the objective function m-file

5.4.2 Running the optimization locally

► 49. *some exercises with interpretation of the results.*

Chapter 6

How to structure your problem

1. example linear tank
2. which folders are expected to exist, what should be in it
3. interface for the functions
4. state space formulation
5. configuration settings for SODA

Chapter 7

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. Within the context of optimization, another typical application of running graphical applications remotely is that you can have the LISA system generate figures showing the progress of the optimization similar to the one in Fig. **??**, which you can then view and manipulate (e.g. zoom, pan) from your local machine. 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 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. 7.1). This will bring up a wizard (Fig. 7.2). Follow the steps outlined in Figs. 7.3–7.5.

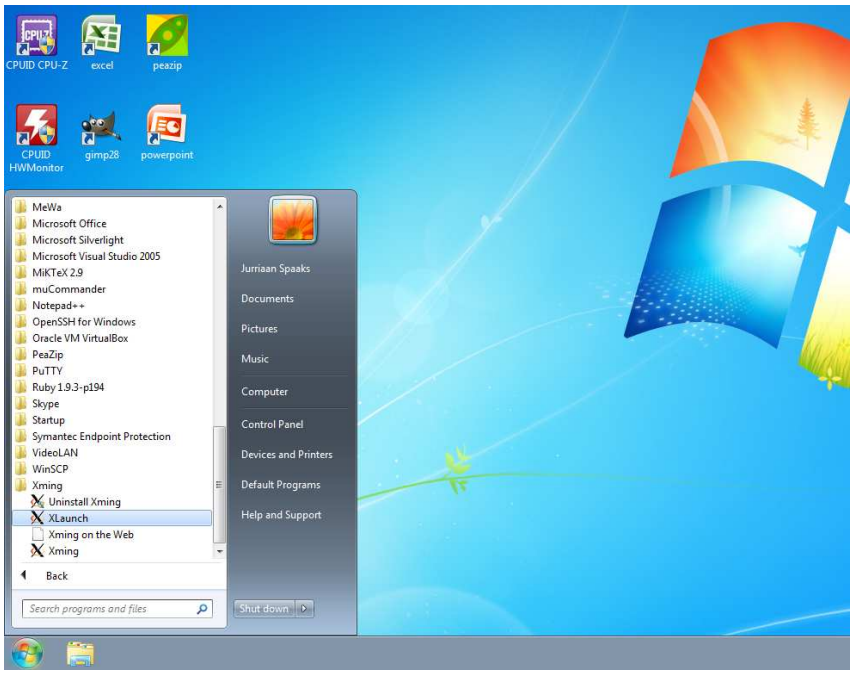

Figure 7.1: Starting XLaunch from the Windows 7 start menu.

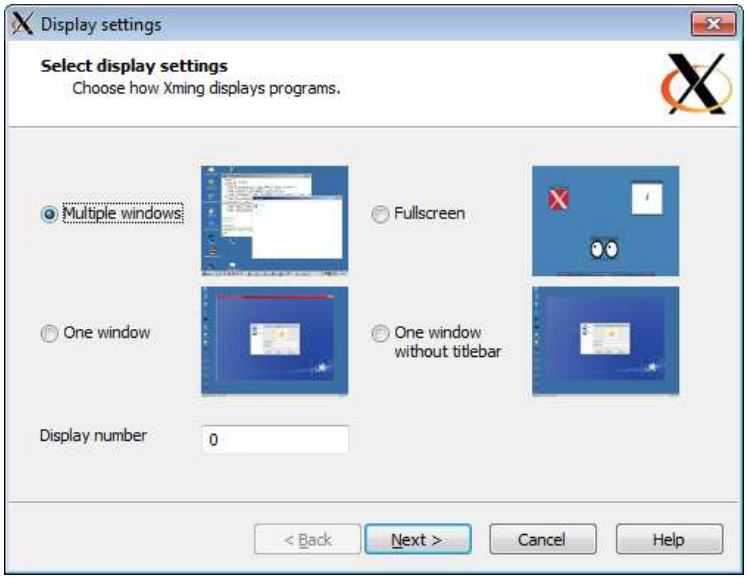

Figure 7.2: The XLaunch configuration wizard page 1.

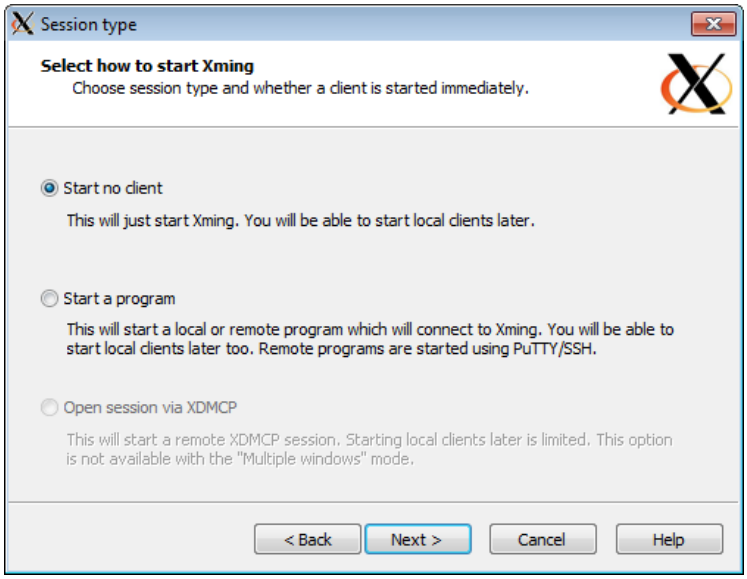

Figure 7.3: The XLaunch configuration wizard page 2.

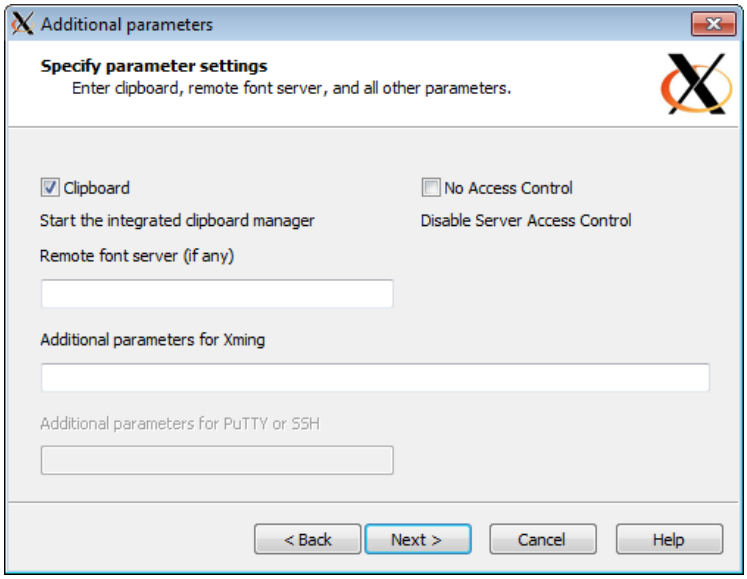

Figure 7.4: The XLaunch configuration wizard page 3.

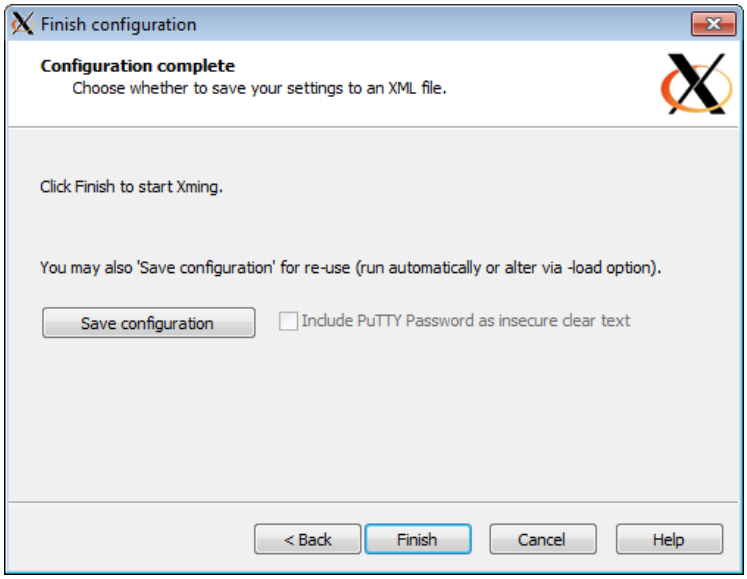

Figure 7.5: The XLaunch configuration wizard page 4.

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.sara.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. 7.6). 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@login4:~$ 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@login4:~$ 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. 7.7.

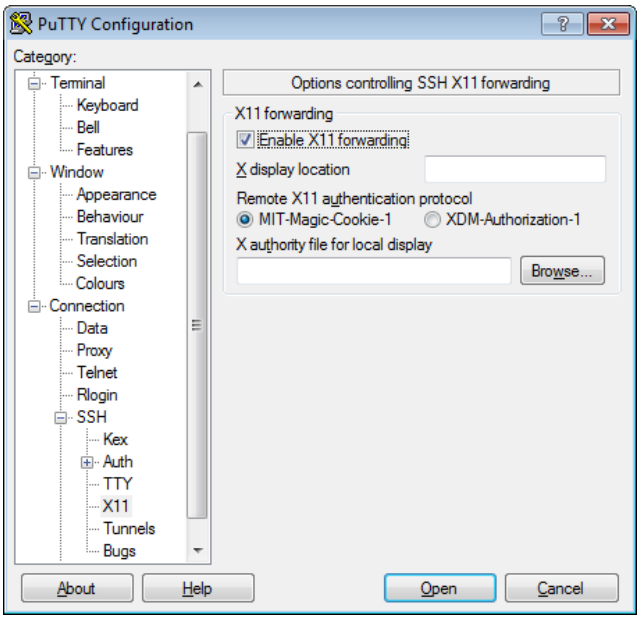

Figure 7.6: Configure PuTTY for X forwarding over SSH.

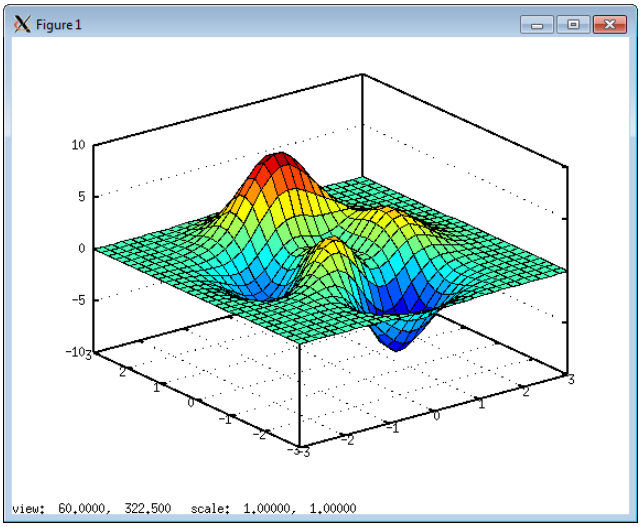

Figure 7.7: After Octave generated this `peaks(30)` figure remotely, the remote system sent it to the local machine over SSH where it was displayed with Xming.

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

```
octave:2> exit

jspaaks@login4:~$ 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@login4:~$ module load matlab
jspaaks@login4:~$ matlab
```

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

```
jspaaks@login4:~$ 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@login4:~$ module load mcr
```

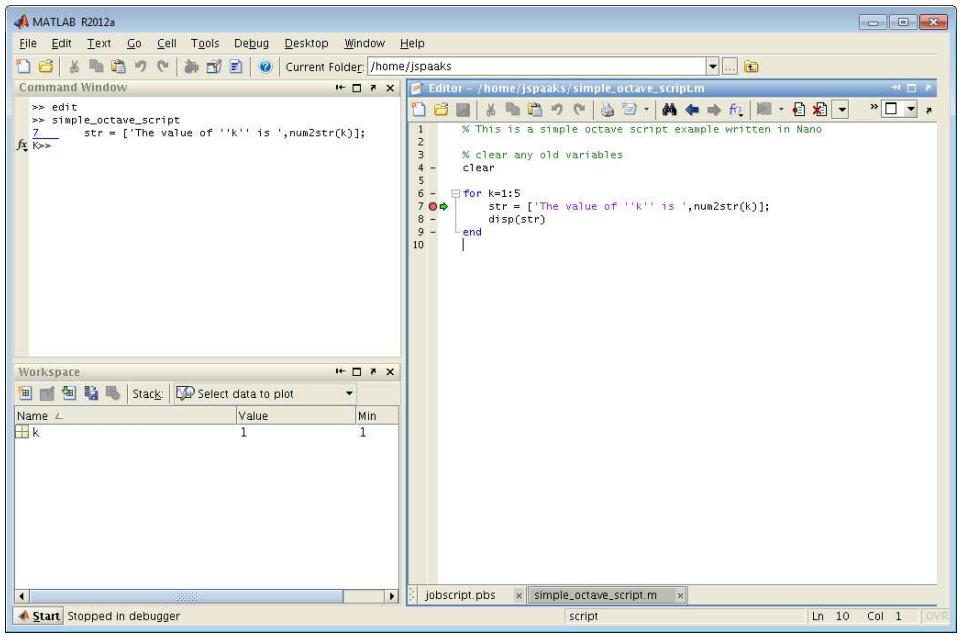

Figure 7.8: Even MATLAB's debugging capabilities can be used remotely!

[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.

Bibliography

Storn, R. and K. Price (1997). Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization 11*, 341–359.

Vrugt, J. A., C. G. H. Diks, H. V. Gupta, W. Bouten, and J. M. Verstraten (2005). Improved treatment of uncertainty in hydrologic modeling: Combining the strengths of global optimization and data assimilation. *Water Resources Research 41*, W01017.

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

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

Vrugt, J. A., C. J. F. Ter Braak, C. G. H. Diks, B. A. Robinson, J. M. Hyman, and D. Higdon (2009). 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.

Index