Linux for scientists

or:

What can I do at the black screen?

Dr. Lennart C. Karssen
PolyΩmica, The Netherlands
l.c.karssen@polyomica.com

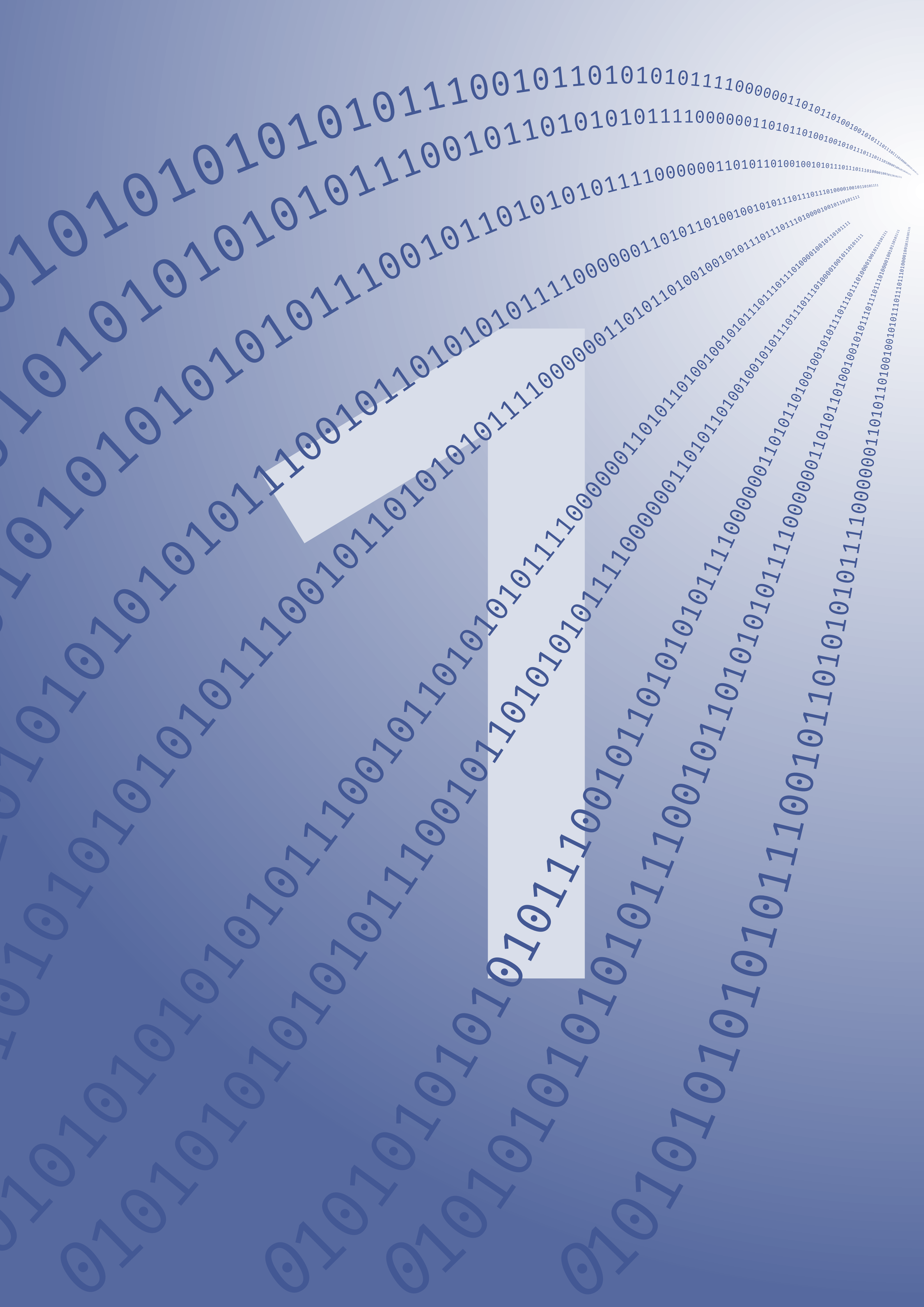
October 2019*

* Git information:
  Hash: c8ba1c9
  Date: 2019-10-01

Contents

List of Tables

List of Tables


1

Chapter 1

Preface

1.1 About this book

This book was initially a collection of lecture notes written for the GE14 "Linux for Scientists" course given at the Erasmus University Medical Centre (ErasmusMC), Rotterdam as part of the NIHES MSc programmes. It focuses on using the Command Line Interface (CLI), if you are using Linux on your desktop you will have noticed that tons of other programs that use a Graphical User Interface (GUI) (i.e. the Firefox web browser) are available also, so this course does not discuss the complete ecosystem of applications for Linux.

This book is split up into several chapters, in roughly the same order as presented in the lectures, ranging from basic file and directory management to Bash scripting and working with the SGE batch queue system. Some exercises are labelled with two or more stars indicating their difficulty level. Some use files and/or tools that are pre-installed on the `epib-genstat.erasmusmc.nl` servers on which this course was initially taught and are therefore less easy to do on other Linux (or Unix-like) systems, but in general these exercises should be quite portable.

Several exercises build on results of previous exercises (e.g. a previously created directory structure, previously copied files, etc.), so it is advisable to work through them in order. Answers to the questions are given at the end of each section or chapter. Remember that there usually are more ways to arrive at the same result[a] and that the solution presented in the answers is not necessarily the best one. Also note that several of the answers contain additional information and tips. Be sure to read those!

A $ at the beginning of a line in the output indicates the command line prompt. You don't have to type it, it should already be visible on the command line after logging in. The symbol ↩ indicates that the command (or the output) is continued on the next line but (in the case of input) should be entered on one line.

[a] People familiar with the Perl scripting language may know the abbreviation TIMTOWTDI (pronounced Tim Toady), which stands for: "There is more than one way to do it".

Please note that this document is under active development. Feel free to contact the author for the latest edition or to report mistakes and typos or other suggestions.

1.2 Acknowledgements

First and foremost: Aaron Isaacs, back in 2010 you asked me if I could teach some Linux/command line tricks to the members of the Genetic Epidemiology group at the ErasmusMC, where I was working at the time. This gave me the final push to set up a practical, hands-on Linux course for scientists.

Cornelia van Duijn, as head of the Genetic Epidemiology Group at the ErasmusMC you allowed me to spend time working on this book. In the course of the next three years the foundations of this book were laid. Thank you for giving me this opportunity.

Other GenEpi group members, Najaf Amin, Maarten Kooyman, Elisa van Leeuwen, Sara Willems, Ayşe Demirkan, Carla Ibrahim-Verbaas, thank you for being part of my first audience, my test users, and for the examples of real-world use cases you provided.

Finally, Yurii Aulchenko, colleague and business partner, your suggestion to make this the first book published under the PolyΩmica umbrella was what made me convert and expand the lecture notes to the book you are now reading.

2

Chapter 2

What is Linux?

Linux, or more precisely GNU/Linux, is an Operating System (OS) usually combined with some additional programs. Other operating systems you might know are Microsoft Windows, Apple macOS and Google's Android.

GNU/Linux basically consists of two parts, the Linux kernel and the GNU tools. A kernel is the piece of software at the heart of each OS, it gives each program the memory it requires whilemaking sure that two programs don't use the same piece of memory at the same time. It allots each of the programs that are running on the computer their time slot on the processor and also takes care of low-level communication between the different peripherals of a computer. For example, if you press a key on the keyboard, it is the kernel that stops the current program, takes time to find out what key was pressed and sends that information to the program you were running. Then the kernel allows the program to continue and process the key your pressed. The Linux kernel was started in 1991[a] as a hobby project of Linux Torvalds, a Finnish computer science student [2, 3]. He published his code under the GNU Public License (GPL), a copyleft licence developed by the GNU project.

In the 1970's the PC (Personal Computer) did not exist yet. Computers were large machines occupying a complete room. Most of them used Unix as OS, augmented by many programs written by the scientist using the computers. Many of these programs were distributed by the authors, but the core Unix programs were owned by companies like AT&T.

The GNU project[b] was started in 1984 by Richard Stallman. The project's goal was to create an OS that was freely[c] available. The GNU tools were written to be compatible with (but not exact copies of) the Unix tools known at the time, hence the meaning of the (recursive) GNU acronym:

[a] A short video that commemorates the 20th anniversary of Linux can be found in Ref. [1].

[b] `http://www.gnu.org`

[c] Note that there are two definitions of "free", free as in freedom and free as in "a free beer". The free Stallman referred to was the former one. As user of the software you are free to do with it what you want. You can modify it, copy it and use it on as many devices as you like. Basically the only restriction is that if you make a change to the source code of the program and distribute that, you'll need to make those changes public as well.

GNU's Not Unix. However, the GNU project never succeeded in writing its own kernel. The rest should not come as a surprise. People using and developing the Linux kernel needed tools to make their OS do something useful, the GNU project needed a kernel, and, well, the rest is history. Nowadays Linux powers more than 50% of the web servers [4], more than 90% of the world's supercomputers [5] and has a small market share on desktop computers. However, most people get in contact with Linux through their mobile phones: Google's Android OS is also based on the Linux kernel (but without the GNU tools).

Because Linux and the GNU tools both use the GPL, anybody can download them, put them together, add their own customisations and redistribute the new set of packages. This is exactly what happened. These sets of packages are called distributions and there are many. Many companies use Red Hat Enterprise Linux[d)], and Ubuntu Linux[e)] and Linux Mint[f)] are used on many home computers and laptops.

Incidentally, Mac OS X is also based on a Unix derivative (called BSD) and as a result many of the things you will learn in this course can also be applied to Apple's OS. You normally don't notice the CLI because it is hidden by the GUI, but it is definitely there. Try looking for the Terminal application.

[d)] `http://www.redhat.com/rhel`

[e)] `http://www.ubuntu.com`

[f)] `http://linuxmint.com`

3

Chapter 3

The basics

To make an OS usable you do not only need a kernel, but, as mentioned in § 2, you also need a set of tools. The most important one is a so-called shell. The shell is a program that allows you to interact with the computer. In fact, it is the part you are looking at most of the time, it is the black screen in which you enter the commands. Most Linux distributions use the Bash shell by default. Examples of other shells are the Z-shell (zsh), the C-shell (csh) and the Korn shell (ksh). Most shells share the same set of basic commands but differ in more advanced functionality. This course assumes you use Bash.

3.1 Logging in and out

ssh

To log on to the server we make use of the Secure Shell (SSH) protocol[a]. On a Linux computer or an Apple Mac you would type

```
ssh username@servername.domain.nl
```

to connect to a server. On Windows the MobaXterm program[b] does the same thing. Putty is a popular alternative, but it lacks certain convenient features, like tabs for multiple connections, etc.

Whichever way you choose to connect to a Linux machine, you will end up with a screen (also referred to as the terminal) with the following text:

```
username@servername:~$
```

This is known as the prompt. It tells you who you are (remember that Linux is a multiuser system), on which computer or server you are (handy when you have multiple connections to different servers) and in which directory. The ~ is a shortcut for your home directory. The final $ denotes the end of the prompt. From here you can start typing commands.

[a] SSH encrypts the communication between your PC and the server. This way a malicious hacker cannot intercept the commands you are typing. This is similar to the https connection to your online banking account in your web browser.

[b] You can download it from http://mobaxterm.mobatek.net/, see § 2.4 on https://epib-genstat.erasmusmc.nl/ for instructions on how to install and configure MobaXterm.

And when you are tired of it all, the commands exit and logout will close the SSH connection to the server.

exit

logout

3.1.1 X11 forwarding: allowing application windows to 'travel' from the server to your PC

For some applications it may be handy to enable so-called X11 forwarding, which allows windows of applications that run on the server to be shown on your local screen. Think, for example, of the plot window in R. Most, if not all tools and utilities discussed in this course do not use X11 forwarding, so feel free to skip this section.

Whether or not X11 forwarding is enabled by default when you log in, depends on how you connect to the Linux server. If you use MobaXterm from Windows this is done automatically, with Putty this is not the case and additional software is required. When connecting from another Linux PC using the ssh command you need to add the -X option in order to enable X11 forwarding:

ssh -X

```
ssh -X username@servername.domain.nl
```

Those who connect from macOS using the ssh command need the -X option as well, but most often also need to install additional software.

To test whether X11 forwarding works, try the following command:

xclock

```
xclock
```

This should show a window with a clock on your local computer. Depending on the speed of the connection between your computer and the server it may take some time (from a few seconds to a minute) for the clock to show up. Note that the window may be hidden below other windows. As long as the xclock application is running, you will not be able to type new commands, in fact, the prompt with the $ symbol is not shown until you close the xclock window. If, for some reason, the $ doesn't show up and neither does the xclock window you can type Ctrl-c to kill the xclock command and return to the prompt.

3.2 Editors

When working on a Linux system two programs are the most important: the shell we just mentioned and your text editor. The better you get to know them both, the more you will gain in terms of efficiency and power over your data. This course focuses on working with the Bash shell, but an editor is needed when writing scripts, README files, etc.

Most Linux systems have a range of editors installed. The most common ones are

emacs

- emacs: A powerful editor, my personal choice. Runs R interactively, similar to Tinn-R on Windows. An Emacs reference card can be found at http://www.ic.unicamp.br/~helio/disciplinas/MC102/Emacs_Reference_Card.pdf, but one is also installed along with Emacs[c].

vim
vi

- vim (or its family members vi and gvim): Another powerful editor. It has two modes, one in which you type your text, the other for commands like open file, save, search/replace, etc. A vi reference card can be found at http://web.mit.edu/merolish/Public/vi-ref.pdf.

nano

- nano A very simple editor, easy to use. However, I strongly recommend learning how to use one of the above.

You might wonder why I encourage you to learn how to use a text editor. You could say “I can stick to Windows’ Notepad for editing my scripts, input files and the like.” And indeed you could. However, that means that you need to transfer every file you want to edit from the server to your Windows PC, and why go through that extra effort? Also, Notepad is very simple whereas Emacs and Vim have a lot of additional features like syntax highlighting for various scripting languages, advanced search and replace options, etc. Moreover, by creating files on two different systems you have to keep an eye on your bookkeeping, “Am I running this script on the latest file, or is that still on my Windows PC?”, as well as make

[c] You can find it by typing `locate refcard.pdf` on the command line. Look for the one simply called refcard.pdf in a directory that has emacs in its name. On my computer it is located in /usr/share/emacs/24.2/etc/refcards/refcard.pdf

sure you won't run into problems with differences in the way Windows and Linux store files (as will be explained in § 4.1).

Both editors have a GUI with buttons and menus as well. If you use MobaXterm you will be able to use these as well. This is especially handy in the beginning when you haven't mastered all the keyboard shortcuts yet (although I urge you to learn those as well). Emacs checks to see if it can open a GUI at start-up (if you want to run the non-GUI version explicitely, then start Emacs with the `-nw` option (which stands for 'no windows'): `emacs -nw filename`). The GUI version of Vim is called `gvim`.

`emacs -nw`

`gvim`

Incidentally, Vim and Emacs can run on Windows as well (in case the keyboard shortcuts of your editor have become ingrained in you spinal cord and you never want to see Notepad again).

Table 3.1 lists some of the basic keyboard shortcuts for Emacs that will get you through this course.

Action	Shortcut
"find" file i.e. open/create a file	`C-x C-f`
save the file	`C-x C-s`
write the file with alternate name (Save as)	`C-x C-w`
exit Emacs	`C-x C-c`
cancel a command	`C-g`
undo	`C-/`
go to beginning of line (Home)	`C-a`
go to end of line (End)	`C-e`
search forward	`C-s`
search backward	`C-r`
use the menu's at the top (File, Edit, etc.)	`F10` or ``M-` ``

Table 3.1: Basic Emacs keyboard shortcuts. Note that `C` stands for the `Ctrl` key, and `M` is the `Meta` key (on modern keyboards you can use either the `Alt` key or the `Esc` key). The combination `C-x` means pressing both the `Ctrl` and `x` key at the same time and ``M-` `` means pressing the `Esc` key and the `` ` `` key (that is the backtick character, usually found on the same key as the ~).

3.3 The structure of Linux commands

In general Linux commands have the following structure:

```
command option(s) arguments
```

Options are keywords that modify the way the command works. Arguments usually indicate what (e.g. which file or directory) the command has to operate upon.

Options are preceded by two dashes. For example, to list all files (i.e. including the hidden ones) on the present directory you can use the following command

ls

```
$ ls --all
```

For those of you that have used the plink tool before, the following command will be familiar. It shows three options, two of which have an argument:

```
$ plink --file inputfile --freq --out newfile
```

Here --file, --freq and --out are options, they modify the default behaviour of plink. inputfile is the argument for the --file option and newfile is the argument for the --out option. In most cases the order in which the options as listed does not matter, as long as you keep the arguments together with their respective options. The command

```
$ plink --freq --out newfile --file inputfile
```

is OK and identical to the previous one, whereas

```
$ plink --out --freq --file newfile inputfile
```

will result in error messages.

Some programs allow short forms of their options as well (this can save you a lot of typing!). Short options consist of a single dash followed by a single letter. For example, the --all option to the ls command can be abbreviated as -a. Consequently, the following two commands are identical:

```
$ ls --all
$ ls -a
```

In most cases the order of the options is not important and multiple single-letter options can be strung together. Again, the following two commands result in the exact same output:

```
$ ls -a -h -l
$ ls -ahl
```

Arguments usually tell the command to operate on certain files or directories:

```
$ ls *.pdf
$ ls /var/log/
```

The first line lists only the files ending in `.pdf` in the current directory (cf. § 3.6.3 for an explanation of the `*`) and the second line lists all files and directories in the directory `/var/log`.

Since a space is used to separate commands, options and arguments, it needs to be 'escaped' by using the \ character when used in a file or directory name, or by enclosing the whole name in double quotes. This means that if you want to make a directory with the name `My scripts are in here` it has to be done in either of the following two ways:

```
$ mkdir My\ Scripts\ are\ in\ here
$ mkdir "My Scripts are in here"
```

Furthermore, it is important to remember that commands and file and diretory names are case sensitive in Linux (unlike in Windows where `C:\Program Files` is equal to `C:\program files`).

One very important feature of the shell is Tab-completion. The concept is very simple: you can use the `[TAB]` key to complete file names and several commands while working on the commands line. This saves an enormous amount of typing! For example, if you have a directory called `MyWork` and you want to use `ls` to list all the files in that directory, simply type

```
ls M
```

and hit the [TAB] key. The shell will try to complete the directory name as far as possible. If MyWork is the only directory that starts with an M it will complete it fully. If, however, you also have a directory called MyPapers, then the result of hitting the [TAB] key will be

```
ls My
```

Hitting [TAB] again will show all possible completions (in this example MyWork and MyPapers). Add a W to what you have typed and hit [TAB] again to complete the directory name.

As we move further through this course you will find that the commands will grow longer and longer. To keep an overview while typing a long command you can insert a \ followed by [ENTER] at any point between a command and its options to continue typing on the next line:

```
$ some_command --option1 \
> --another-option some_argument \
> --yet-another-option another_argument
```

Note that the > should not be typed. It will is inserted by the shell to inform you that you haven't finished the command yet.

3.3.1 Exercises

Exercise E3.1 Long and short options

In this section we saw that the ls command accepts options in both long and short form.

a) What is the difference between running

```
$ ls --all
```

and

```
$ ls -all
```

Exercise E3.2 Working with the command history

With so many commands to remember, working on the command line can seem to be daunting. Luckily there is a sort of memory. A history of your past commands is saved and ready for you to use.

history

a) Use the `history` command to list the recent commands you typed. Use the up and down arrow keys to cycle through some of your recent commands. Once you have found an interesting command you can try to edit it and run it again. Note that the history does not remember in which directory you were when you executed a command.

b) Using the up and down arrow keys is fine if you are looking for a recently used command. For commands that are higher up in the history list this becomes too cumbersome. For those situations you can search through the history with the `Ctrl-r` (reverse search) shortcut. Simply hit `Ctrl-r` (i.e. press and hold the `Ctrl` key and then press the `r` key and release both) and start typing some part of the command that you remember. For example, if you are looking for the command

```
$ ls /storage/imputations/ERF3_HM2_Mach_2010.10.01/Results/
```

that you typed some time ago, you can hit `Ctrl-r` followed by typing `ERF3`. However, if after typing the above command you had typed `mkdir ERF3` than this entry would show up first. Hit `Ctrl-r` again to cycle through all entries that contain `ERF3` until you reach the one you are looking for or continue typing your search criterion until it becomes unique. Once you have found the command you want to reuse you can either press the `Enter` key to rerun the command or change the command according to your whishes. To cancel your search simply press `Ctrl-c`.

3.4 Managing your account

By now you have hopefully logged into a Linux system. Since many Linux servers can be directly accessed from the Internet it is important to keep your account information (user name and password) secret. Changing your password regularly (at least once a year) is a good security measure. To change your password use the `passwd` command.

passwd

In order to make sure that no single user can use up all disk space (accidentally or not), every user on a server has a certain disk quota allotted for his/her files. You can check your quota status with the `quota` command:

quota

quota -s

```
$ quota -s
Disk quotas for user lennart (uid 1305):
  Filesystem  blocks   quota   limit   grace   files   quota   limit   grace
    /dev/md0    206G    250G    255G            4800       0       0
```

The interesting columns are columns 2, 3, 4 and 5 (you can forget about the last four columns). The `blocks` column shows your current usage (206 GB in this case), the `quota` column shows your maximum. You will be notified by e-mail if you exceed it.

After exceeding this maximum disk space you will have 7 days of 'grace' (the number of days left is noted in the `grace` column), in which you can still use some more space (up to the value in the `limit` column, 255 GB in this example). Once the grace period has expired you will not be able to create any more files. This can even prevent you from logging in!

3.5 Getting help

Most human beings won't be able to remember all the options for all the commands they use on a regular basis. And what about the commands you only use once in a while? How do you find out which options are available, what the actual use of a certain command is, what does it expect as input, etc.?

To remind you of the most commonly used options as well as expected input and output most commands have a `--help` option. This is the (truncated) output for `ls`, for example:

```
$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort.

Mandatory arguments to long options are mandatory for short options too.
  -a, --all            do not ignore entries starting with .
  -A, --almost-all     do not list implied . and ..
      --author         with -l, print the author of each file
  -b, --escape         print C-style escapes for nongraphic characters
      --block-size=SIZE use SIZE-byte blocks. See SIZE format below
  -B, --ignore-backups do not list implied entries ending with ~
```

The first line explains the usage of the command, i.e. what to type on the command line, followed by a short two-line description. After that a list of most of the options, mentioning both the long and the short form (if available).

For more detailed information most commands also have a manual page which is shown on screen by the `man` command. This is the start of the man page for `ls`:

man

```
LS(1)                    User Commands                    LS(1)

NAME
     ls - list directory contents

SYNOPSIS
     ls [OPTION]... [FILE]...

DESCRIPTION
     List information about the FILEs (the current directory by ↩
         default).
     Sort entries alphabetically if none of -cftuvSUX nor --sort.

     Mandatory arguments to long options are mandatory for short options
     too.

     -a, --all
          do not ignore entries starting with .
```

```
    -A, --almost-all
           do not list implied . and ..

    --author
           with -l, print the author of each file

    -b, --escape
           print C-style escapes for nongraphic characters
```

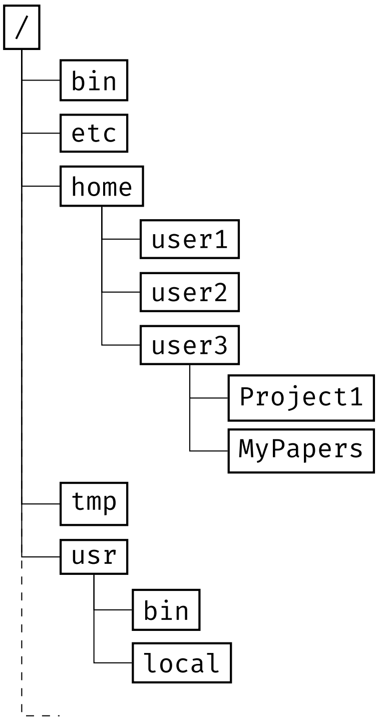
To browse through the man pages use the arrow keys, PgUp, PgDn, etc. To quit the man page browser use the `q` key[d].

And, of course, the Internet is a great resource. If you are struggling with a command Google Is Your Friend (GIYF).

3.6 Working with files and directories

On Windows and GNU/Linux systems files are ordered in directories. In contrast to Windows with its drive letters (`C:`, `D:`, etc.), directories on a Linux system are ordered in a single tree starting at the root, indicated by `/`:

[d] In fact, the program used to view these man pages is `less`, a so-called pager that will be discussed in more detail in §3.8



As mentioned in § 3.3, the `ls` command is used to list the contents of directories. If entered without arguments it simply lists the contents of the present working directory (pwd). The following options will get you by most of the time: `ls`

- `-a` list all files, i.e. including hidden files[e)] `ls -a`
- `-l` long list, this listing shows the time and date a file was modified, its size, permissions and the user and group of the owner of the file or directory `ls -l`
- `-h` usually used in combination with `-l`, it shows the file size in ’human readable’ format, i.e. in MB, GB, etc. instead of in bytes `ls -h`
- `-d` use when the arguments are only directories, not files, list information on the directory itself, not its contents (see below) `ls -d`
- `-t` sorts the entries by modification time `ls -t`

[e)] On a Unix system all files and directories whose name starts with a dot are hidden.

If the argument of `ls` consists of only (one or more) directories, `ls` shows only the information of the files in that directory, not of the directory itself. The `-d` option changes this behaviour:

```
$ ls dir1 dir2
dir1:
dir1_2 file1 file2 file3

dir2:
file11 file12 file13
$ ls -l dir1 dir2
dir1:
total 16
drwxr-x--- 2 lennart genepi 4096 2011-10-12 11:09 dir1_2
-rw-r----- 1 lennart genepi 4 2011-10-12 11:12 file1
-rw-r----- 1 lennart genepi 7 2011-10-12 11:12 file2
-rw-r----- 1 lennart genepi 614 2011-10-12 11:13 file3

dir2:
total 12
-rw-r----- 1 lennart genepi 201 2011-10-12 11:13 file11
-rw-r----- 1 lennart genepi 603 2011-10-12 11:14 file12
-rw-r----- 1 lennart genepi 340 2011-10-12 11:14 file13
$ ls -d dir1 dir2
dir1 dir2
$ ls -ld dir1 dir2
drwxr-x--- 3 lennart genepi 4096 2011-10-12 11:09 dir1
drwxr-x--- 2 lennart genepi 4096 2011-10-12 10:59 dir2
```

The output of the `-l` option also warrants some more explanation. Consider the following:

```
lennart@server:~/ErasmusMC$ ls -l
total 712
drwxr-xr-x  6 lennart genepi   4096 2011-09-29 18:57 Articles
drwxr-x---  6 lennart genepi   4096 2011-07-06 09:07 Conferences
drwxr-x---  5 lennart genepi   4096 2011-08-29 18:18 Courses
drwxr-x---  2 lennart genepi   4096 2010-11-29 19:26 Graphics
drwxr-x---  7 lennart genepi   4096 2011-09-27 18:26 MeetingNotes
-rw-r-----  1 lennart genepi  50231 2011-09-02 14:57 notes.org
```

```
drwxr-xr-x  3 lennart genepi   4096 2011-05-23 16:21 PaperReviews
drwxr-x--- 18 lennart genepi   4096 2011-09-27 18:14 Projecten
drwxr-x---  4 lennart genepi   4096 2011-06-21 01:21 R-dev
drwxr-x--x 14 lennart genepi   4096 2011-09-02 14:58 ServerBeheer
drwxr-x---  2 lennart genepi   4096 2010-07-28 13:44 snipextract
-rw-r--r--  1 lennart genepi    753 2011-05-12 18:07 todo.txt
```

The output consists of eight columns, the last one being the file or directory name. Column one shows the permissions (which will be discussed in § 3.10.2), columns three and four show the owner and group of the file (§ 3.10.1). Column five shows the size of the file in bytes (unless the `-h` option is also used). Finally, columns six and seven show the modification time and date, respectively.

3.6.1 Directories

pwd

Each user has a home directory for his/her own files. All home directories are subdirectories of `/home` and are named after the user's user name, i.e. `/home/your_username`. After logging in you will find yourself in your home directory. You can check this by running the `pwd` command, which shows your present working directory (pwd). The tilde (~) is used as a shortcut for the path to your home directory.

cd

To move into another directory use the `cd` command, which stands for *c*hange *d*irectory. The `cd` command can be followed by either a relative path or an absolute path. An absolute path starts from the root (/) directory, e.g. `/home/lennart/Programming` or `/tmp`. A relative path uses the pwd as starting point to go up or down the directory tree and consequently doesn't start with a /. Take a look at the following examples and notice that here the whole shell prompt is shown (instead of just $) to indicate how the shell prompt changes to show your pwd:

```
lennart@server:~$ cd ErasmusMC/
lennart@server:~/ErasmusMC$ cd Courses
lennart@server:~/ErasmusMC/Courses$ cd /tmp
lennart@server:/tmp$ cd /home
lennart@server:/home$ cd /
lennart@server:/$ cd ~
```

```
7 lennart@server:~$ cd ErasmusMC/Conferences/2011
8 lennart@server:~/ErasmusMC/Conferences/2011$ cd ..
9 lennart@server:~/ErasmusMC/Conferences/$
```

The paths used in lines 3, 4, 5 and 6 are absolute paths, the others are relative paths. Line 8 shows a special kind of relative path, the `..`, which means the parent directory (i.e. one level up in the tree). It can be used just like any other relative path, so `cd ../..` means go up two levels in the directory hierarchy. The pwd also has a shortcut: `.` (i.e. a single period), so the command `cd .` does nothing, since it just means "go to the directory you are already in". More interesting use of the `.` shortcut is to explicitly indicate the pwd, for example when copying:

```
$ cp data_chr*.dat ./Project1/data/
```

which means "copy the files starting with `data_chr` and ending in `.dat` to the directory `Project1/data`, which is a subdirectory of the pwd". Note that the following statement is equivalent:

```
$ cp data_chr*.dat Project1/data/
```

Another use case, running scripts from the pwd will be explained in Chapter 5, page 85.

`mkdir`
`rmdir`

Directories can be created by the `mkdir` command and removed with the `rmdir` command. Both accept the name of one or more directories as an argument. Note that the `rmdir` command only removes empty directories, and will give a warning if the directory is not empty. Use the `-r` option of the `rm` command to recursively remove a directory and its contents (the `rm` command is explained in more detail in § 3.6.2).

3.6.2 Copying, moving, removing

`cp`

Managing files and directories revolves around the concepts of copying, moving and removing. Files and directories can be copied using the `cp` command, which needs (at least) two arguments. The last one is always the destination, the ones before that are the source files (or directories).

For example, to copy file1 to a new file with name file2 type:

```
$ cp file1 file2
```

This copies files file1 and file2 to the directory dir1 (which must already exist):

```
$ cp file1 file2 dir1
```

To copy file1 and file2 to the directory dir1, which is located at the same level in the directory tree as our pwd type:

```
$ cp file1 file2 ../dir1 #
```

cp -i

Be careful, by default the cp command doesn't warn you when a file with the same name exists in the destination, it simply overwrites it! If you add the -i option when using the copy command you will be asked for confirmation if the destination already exists.

cp -r

The most used option for the cp command is the -r option, which allows one to copy directories and files recursively. If you have tried to use a directory as the first argument you have already noticed the following error message:

```
$ cp dir1 dir2
cp: omitting directory `dir1/'
```

The -r option fixes this, as we can see with the ls command.

```
$ cp -r dir1 dir2
$ ls -ld dir1 dir2
drwxr-x--- 2 lennart genepi 4096 2011-10-12 10:57 dir1
drwxr-x--- 2 lennart genepi 4096 2011-10-12 10:59 dir2
```

mv

Moving a file (or directory) is very similar to copying it, the command mv is followed by at least two arguments: the source and the destination.

```
$ mv file1 dir1  # Move file1 into directory dir1
$ mv file1 file2 dir2 # Move file1 and file2 into dir2
```

Unlike cp, mv works recursively by default:

```
$ mv dir1 dir2
```

mv -i

moves dir1 (and it contents) into dir2, assuming dir2 exists. If dir2 does not exist, dir1 will effectively be renamed into dir2. The same holds for files. If you move one file and the destination file does not exist, the effect of the move is that the file is renamed. Like the cp command, mv also overwrites an already existing destination file without warning. As in the case of cp use mv -i to get a prompt before overwriting.

rm
rm -r
rm -i

And now we come to the more dangerous part: removing files and directories. It is very important to realise that Linux expects people to know what they are doing. Consequently, there is no such thing like the Trash folder on Windows, once a file is deleted, it is gone!! The command for removing files is which accepts one or more file names as arguments. Like cp rm does not work recursively by default, and like with cp, the -r option enables it. Be careful as -r will also remove directories. Similarly, the -i option asks for confirmation.

```
$ rm file1 file2 # file1 and file2 are deleted
$ rm dir1        # Won't work, whether dir1 is empty or not
rm: cannot remove `dir1': Is a directory
$ rm -r dir1
```

The last line removes dir1 and all the files and subdirectories within it!

3.6.3 Wildcards

Wildcards make it easier to manage multiple files at a time. These are the two wildcards and their use:

*: replaces one or more characters
?: replaces one single character

The following examples show how to use wildcards.

- Show all files in the pwd:

  ```
  $ ls *
  ```

- Show all files ending in pdf in the pwd:

  ```
  $ ls *.pdf
  ```

- Show all files that start with chr and end with .dat:

  ```
  $ ls chr*.dat
  ```

- Show all files that start with chr, have two characters in between and end with .dat:

  ```
  $ ls chr??.dat
  ```

- Copy all files ending in .dat to a Backups directory (which is located in the pwd):

  ```
  $ cp *.dat Backups/
  ```

Note that when using cp or mv wildcards can only be used in the first argument (the source), not in the second (the destination). This makes sense, of course, since most of the times you'd like to copy (or move) a set of files to one directory and not one file to a set of directories.

Using wildcards, especially in combination with the rm command can be dangerous. Before you know it you type

```
$ rm * old.pdf # DANGEROUS, DON'T DO THIS!
```

instead of

```
$ rm *old.pdf
```

In the last case you delete all the files ending in old.pdf, whereas in the first case all files in the pwd are deleted first and then rm tries to delete the file old.pdf (which it probably can't find).

3.6.4 Exercises

Exercise E3.3 Some file and directory basics

In this exercise you will learn to work with directories, how to create them, how to remove them and how to move from one to the other. Don't forget to use Tab-completion as much as possible.

a) Go to your home directory.

b) Create a directory called `LinuxCourse`.

c) Go into that directory and create two other directories called `tmp2` and `tmp3`.

d) Go to the `tmp2` directory.

e) Go to the `tmp3` directory in one step.

f) Go to back your home directory (again in one step).

g) What is the shortcut to go to the previous directory? Use it.

h) What is the absolute path of your present directory? Which command will print it on the screen for you?

i) Go to the root of the file system (/) and list the files and directories there.

j) Go into the `/tmp` directory and list its contents (although the contents will not be very interesting).

k) Use a relative path to go back to the `LinuxCourse` directory (in one step of course).

l) Check if the two directories `tmp2` and `tmp3` are still there and then remove them.

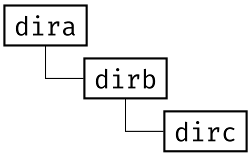
Exercise E3.4 Copying files

What is wrong with the following command (try to think before typing it in and trying the command)?

```
$ cp file1 file2 file3
```

Exercise E3.5** Creating a directory tree

a) Find out how to create a “branch” of directories in one command. For example, make the following directory structure (starting in the LinuxCourse directory):



b) Standing in the parent directory of dira how would you remove these three directories?

Exercise E3.6 Getting information on files and directories

The ls command lists files and directories (and their properties).

a) what is the size of a file with the name vmlinuz followed by some numbers in the directory /boot[f]?

b) Go to the directory /var and list the names of the user and group that own the directories

```
mail
crash
local
```

Exercise E3.7** The dangers of wildcards

Wildcards in combination with rm can have devastating results!

a) Without entering the following commands (watch out, doing so is **EXTREMELY DANGEROUS!!**), can you tell the difference in effect that they have?

```
rm -r *~
rm -r * ~
```

[f] A vmlinuz file is the Linux kernel (see § 2).

3.7 Transferring files from one Linux machine to another

scp

Files can be copied to and from the server with the scp command[g]. A copy action looks just like a regular cp command:

```
$ scp source destination
```

but with scp either source or destination consist of a username@server part followed by a colon and the file(s) to copy from or to, respectively. For example, the command

```
$ scp local_file ↩
   your_username@server.domain.nl:dir/on/the/server/
```

copies a local file to the server and puts it in the specified directory. Copying a file from the server to your own computer works similarly:

```
$ scp your_username@server.domain.nl:~/path/to/the/file ↩
   ~/a/local/dir/
```

scp -r

When copying directories including the files they contain the -r option must be used:

```
$ scp -r some_dir/ ↩
   your_username@server.domain.nl:the_target_directory/
```

rsync

rsync -azP

For transfers of many and/or large files the rsync command is better suited because if something goes wrong and the transfer is aborted[h], rerunning the scp command will start from scratch again. rsync is much smarter about these cases and will transfer only (parts of) files that have not been sent yet. An rsync command looks just like an scp command:

```
$ rsync -azP source destination
```

[g] This works also when copying to and from an Apple computer with Mac OS X.

[h] as mentioned in Chapter 6 tasks taking more than 10 minutes of processor time will be killed unless the batch queue system is used.

The -a option preserves file properties like dates, owner (if possible) etc. and sets the recursive option for directories. The -z option turns on compression of the data (which can decrease transfer times for some files) and the -P option adds progress information and keeps partially transferred files. As with scp either source or destination can be a remote location specified as username@servername:path/to/files.

3.8 Pagers, or how to look at the contents of a file

In section 3.2 we discussed how to use an editor on the server to edit your files. Sometimes, however, you are only interested in seeing the contents of a file, without the need to edit it. In such a case loading the file into an editor is of course an option, but there are faster options.

cat

The simplest way to display the contents of a file is using the cat command:

```
$ cat timings.dat
nids t_1 t_2 t_3
50 0.5 1.4 4.63
100 0.62 0.21 18.19
200 0.9 0.4 10
500 2.4 5.19 171.09
1000 9.08 54.93 1653.25
1500 19.9 320.12 7201.39
2000 36.25 1086 18500.61
2715 74.95 2181.3 31432.52
```

Although cat is simple and straightforward it has some serious limitations. If the contents is larger than the size of the screen you'll need to scroll back and for files that are larger than the terminal's buffer size you wouldn't be able to scroll back all the way to the top.

more
less

To remedy this programs come to the rescue: more and less are the pagers that can be found on any Linux system. These programs are called pagers, because they allow the user to view the contents of the file screen by screen i.e. page by page. To view a file called longfile using either of these pagers simply give it as an argument:

```
$ more longfile
```

Use the space bar to scroll screen by screen and the q key to exit.

zless

Usually less is to be preferred because it allows one to easily browse up and down, as well as to search in the contents[i]. Again use the q key to exit less. In § 3.9 you will learn how to deal with compressed (zipped) files. To read the contents of a gzipped file it is not necessary to unzip it first, simply use the zless program.

head

tail

Two other commands also greatly help with getting a quick glimpse of the contents of a file: head and tail[j]. Without additional options these commands show the first or the last ten lines of a file, respectively. A different number of lines can be given as argument to the -n option:

```
$ head -n 1 timings.dat
nids t_1 t_2 t_3
$ tail -n2 timings.dat
2000 36.25 1086 18500.61
2715 74.95 2181.3 31432.52
```

3.8.1 Exercises

Exercise E3.8 Working with less

a) Open the man page of less.

You can browse through the man page with the up and down arrows, [PgUp], [PgDn], etc. but if you have an idea of what you are looking for it is easier to search for it. Searching in the forward direction is done by

[i] The man command uses less to display the man page on screen.

[j] The R language has similar functions: head() and tail().

pressing the / key, then type your search term and hit [ENTER]. Hitting the keys / [ENTER] key repeatedly will go to the next hit, etc.

b) Which key is used to search backward?

Exercise E3.9 ”Seeing a file grow”

Imagine that you have an analysis that takes a long time to finish but writes some output to a file at every step (e.g. a ProbABEL [6] run, which writes the outcome for each SNP on a new line). To follow progress you could open the file every once in a while to see where it is, but once the file is loaded (in `less`, `more` or even your editor), the output would continue to grow and you would need to reload the file constantly for the updates to arrive. Another way would be to run `tail` every few minutes to only show the last few lines[k].

To save you time, `tail` has an option that follows the contents of a file and shows you each line that is appended. Which option is that?

3.9 Using compressed archives like `.zip` and `tar.gz` files

On a Linux system the following file formats are commonly used for compressed archives:

`.zip` A regular zip file, also recognised on Windows without the need for extra software.

`.gz` A compression format that only contains a single file.

`.tar.gz` A gzipped tar archive, which can contain multiple files. This format is the most common.

[k] Incidentally, the way to do this automatically is via the `watch` program.

3.9.1 zip

zip

unzip

Zip files are created and extracted using the zip and unzip commands respectively. In the following example the directory MyProjectDir (including every file and subdirectory because of the -r option) is first zipped into a file called my_archive.zip. Subsequently, the original directory is removed, followed by the unzipping of the archive to restore the directory and its contents.

```
$ zip -r my_archive.zip MyProjectDir
updating: MyProjectDir/ (stored 0%)
updating: MyProjectDir/testfile1 (stored 0%)
updating: MyProjectDir/testfile3 (deflated 54%)
updating: MyProjectDir/testdir1/ (stored 0%)
updating: MyProjectDir/testdir1/testdir2/ (stored 0%)
updating: MyProjectDir/testfile2 (stored 0%)
updating: MyProjectDir/testdir3/ (stored 0%)
$ rm -r MyProjectDir
$ unzip my_archive.zip
Archive: ../my_archive.zip
   creating: MyProjectDir/
 extracting: MyProjectDir/testfile1
  inflating: MyProjectDir/testfile3
   creating: MyProjectDir/testdir1/
   creating: MyProjectDir/testdir1/testdir2/
 extracting: MyProjectDir/testfile2
   creating: MyProjectDir/testdir3/
```

3.9.2 gzip

gzip

gunzip

zless

To create a Gzipped .gz file use the gzip command and to unzip a .gz file use gunzip. The gzip compresses each file you specify as an argument separately, unlike zip, which stores all files in one .zip file. Furthermore, it removes the original uncompressed file. Similarly, gunzip decompresses the file(s) you give as arguments, but removes the compressed ones. To read gzipped text files use zless (see also § 3.8).

3.9.3 tar

tar

To circumvent Gzip's limitation of compressing each file individually, most people use the tar command. With tar multiple files and directories can be stored in a single file, and it accepts additional options for compression. The following command creates a compressed archive of files, similar to the zip example above.

```
$ tar -czf my_archive.tar.gz MyProjectDir/
```

Here, the -c option stands for "create", the -z option gzips the file and -f specifies the filename of the archive. Extracting a tar.gz archive is very similar:

```
$ tar -xzf my_archive.tar.gz
```

The -x options stands for 'extract'.

3.9.4 Exercises

Exercise E3.10 Untar-ing an archive

For some of the exercises some pre-made files will be needed. A compressed archive of the files can be found in /tmp/exercises_linux_course.tar.gz.

a) Copy the file to your LinuxCourse directory and extract it.

It is considered good practice that a tar.gz file always contains a directory with the same name as the tar.gz file itself. This way, when someone extracts it, (s)he doesn't end up with files lying around in the directory where the extraction was done.

b) Check that the extraction has completed successfully by listing the contents of the present directory.

3.10 File ownership and permissions

Linux is a multi-user OS, which means that several people can access the system at the same time. As result it is important for the OS to be able to tell who is the owner of a certain file or directory as well as who else might have access to that file or directory.

3.10.1 Ownership

whoami

Every user has his or her own user account with a user name given by the system administrator. To find out your user name type the `whoami` command. Furthermore, each user is also assigned to one or more groups. The first group to which a user is assigned is his/her default group. The command `id` shows your user ID (uid), group ID (gid) and all other groups you are a member of:

```
$ id lennart
uid=1305(lennart) gid=10001(genepi) ↩
    groups=10001(genepi),100(users),10009(wikiusers),10010(gvnl),10011(svn)
```

Here you see that my user name is “lennart” and my default group is “genepi”. The names that follow are other groups that I am a member of. The numbers are not too important at this moment, suffice it to say that each user or group ID has a unique number associated to it.

To see the user and group ownership of a file use `ls -l`:

```
$ ls -l
total 3529936
drwxr-xr-x   4 lennart genepi       4096 Nov 19  2010 BigGrid
drwxr-xr-x   4 lennart genepi       4096 Aug 22 18:01 bin
-rw-r-----   1 lennart genepi       1612 Oct 10 01:48 course_desc.txt
drwxr-x---   6 lennart genepi       4096 Jul 12 17:38 Courses
drwxr-x---   3 lennart genepi       4096 Nov 23  2010 CUDA
-rw-rw----   1 lennart gvnl      2047780 May  4 15:59 M34d.zip
drwxr-x---  34 lennart genepi       4096 Aug 24 17:41 Packages
drwxr-x---  11 lennart genepi       4096 Oct  6 10:56 Projecten
drwxr-xr-x   4 lennart genepi       4096 Aug 30 15:56 public_html
drwxr-x---   7 lennart genepi       4096 Mar 17  2011 R
drwxr-x---  16 lennart genepi       4096 Oct  8 01:40 Rlibs
```

```
drwxr-xr-x   5 lennart genepi       4096 Mar  2  2011 Scripts
drwxr-x---   5 lennart genepi       4096 Mar 28  2011 ServerMaintenance
drwxr-x---   2 lennart genepi       4096 Aug 29 13:26 SGE-test
drwxr-xr-x   8 lennart genepi       4096 Dec 13  2010 TeX
drwxr-xr-x  17 lennart genepi       4096 Sep 30 16:24 tmp
drwxr-x---   3 lennart lennart      4096 Jun 24  2010 X-chromosome
```

Here you see that each file or directory is owned by me (column three) and that the associated group for each of them is “genepi” (column four). Only the file Z34d.zip has a different group: “gvnl”. Why this is important will be explained in the next chapter.

3.10.2 Permissions

Closely related to the concept of ownership is the concept of permissions. Each file or directory has a set of permissions associated with it, that indicate who can read, write and/or execute the file. These are indicated by the letters r, w and x, respectively. In its simplest form these three permissions can be set for the “user”, the “group” and “others”[l]. The meaning of the read permission is obvious. Write permission means that the file or directory can be changed, moved and deleted. The execute permission has different meanings for files and directories. For files it means that Linux will try to run the file like a program or script. This will be discussed in more detail in Chapter 5 on writing Bash scripts. For directories the execute permission means that the user, group or others can access that directory (via cd for example).

Let’s take a look again at some ls -l output which shows the permissions on the first column:

```
$ ls -l
total 3529936
drwxr-x---   5 lennart genepi       4096 Mar 28  2011 ServerMaintenance
drwxr-x---   2 lennart genepi       4096 Aug 29 13:26 SGE-test
drwxr-xr-x   8 lennart genepi       4096 Dec 13  2010 TeX
drwxr-xr-x  17 lennart genepi       4096 Sep 30 16:24 tmp
```

[l] For more fine-grained control of permissions the setfacl and getfacl can be used to read and set ACLs (Access Control Lists) on files or directories. The usage of these tools falls beyond the scope of this course.

```
drwxr-x---   3 lennart genepi       4096 Aug 24 17:41 Packages      7
-rw-rw----   1 lennart gvnl      2047780 May  4 15:59 M34d.zip      8
drwxr-x---  11 lennart lennart      4096 Jun 24  2010 X-chromosome  9
```

The first column consists of four blocks. The first block is one character wide and has a `d` if the entry is a directory. Then follow three blocks of three permissions `r`, `w` and `x`, one for the owner, one for the group and one for all other users. If one of the permissions is not set it is indicated by a `-`.

So in this listing you see mostly directories, except for the entry `M34d.zip` for on line 8. This file has the permissions read and write for the owner as well as the group (`gvnl`), but other users have no access to the file. In contrast, the `TeX` directory can be accessed and read by everybody, but only the owner can write to it.

By default the permissions on a file are `rw` (read and write) for the owner, `r` (read-only) for the group and others. This means that everyone can read the file.

chmod

Permissions can be changed with the `chmod` command. Use of this command falls beyond the scope of this course (although it will briefly show up in Chapter 5).

When copying files from a Windows machine it is possible that the permissions are set to `rwx` for all three groups. This is because the Linux/Unix permission scheme is different from the way Windows handles file permissions. You can safely change the parameters to the default values (i.e. `rw` for the user, `r` for the group and none for others) using the following command:

```
$ chmod 640 my_file_from_windows
```

3.11 Process management

Every program or command that you run on a Linux machine starts one or more processes. In this section you will learn some basic commands to manage them. Processes can be killed, sent to the background and be brought back to the foreground.

3.11.1 Exercises

Exercise E3.11 Listing and killing processes

To find out whether the server is busy, you can use the programs top and htop that show dynamically updated lists of processes (usually sorted by processor (also known as Central Processing Unit (CPU)) usage).

htop

a) Start the htop program (top basically does the same, although less colourful). On the screen you can see that the server has 12 CPUs. The amount of memory used is listed as well. In the lines below a list of processes is shown, sorted by CPU usage. By pressing the u key you can select a user and only see his/her processes. Other options are available as well, use the h for help. To return to the command line, press q.

While htop is fun to watch, sometimes you simply want to see which processes you have running. For example, you have just started a large analysis, but discovered you forgot to add a certain piece of information. Instead of letting the current analysis program run, using a lot of processing power while you know you will throw away the results anyway, you can use the ps command to find the process and the kill command to kill it.

kill

ps

b) Run ps -fu username (with your own user name) to find out which processes you have running. Here is some example output:

```
$ ps -fu lennart
UID      PID PPID C STIME TTY    TIME CMD
lennart 4633 4631 0 17:49 ?  00:00:00 sshd: lennart@pts/8
lennart 4634 4633 0 17:49 pts/8 00:00:00 -bash
lennart 5841 31983 0 18:50 pts/6 00:00:00 sh jobscript.sh 1 100000
lennart 5842 5841 63 18:50 pts/6 00:00:00 /usr/lib64/R/bin/exec/R --slave
lennart 5844 4634 0 18:50 pts/8 00:00:00 ps -fu lennart
lennart 29273 29271 0 10:51 ? 00:00:00 sshd: lennart@pts/2
lennart 29274 29273 0 10:51 pts/2 00:00:00 -bash
lennart 31982 31980 0 13:25 ? 00:00:01 sshd: lennart@pts/6
```

```
11 lennart 31983 31982 0 13:25 pts/6 00:00:00 -bash
```

In this example I have nine processes running. The ones in lines 3, 8 and 10 are my connections to the server (i.e. three in total). For each connection one SSH process is started (lines 4, 9, 11), which in turn starts the Bash shell so I can type my commands. In line 5 you see a Bash script that is running, and in line 6 you can see I am running R. Line 7 shows the process of the ps command itself.

In this output, the column PID is an important one. Each process gets a unique process ID, or PID. It is this PID that can be used to kill a process. If, for example, I want to kill the R process, I simply type

kill

```
$ kill 5842
```

and R stops. You probably will not do this very often, but every once in a while it can be handy[m].

Exercise E3.12** Sending processes to the background (and getting them back)

By default, a program that you start on the command line is run in the foreground. This means that the prompt will not reappear until the program has finished. So you will have to wait before you can type a new command. If you don't want to wait, you can type Ctrl-z to suspend the program. A suspended program does nothing. It simply waits for you to bring it back to life (i.e. to the foreground or the background) again.

a) Start R and type one or two simple commands like

```
> a <- "hello"
> ls()
```

Now suspend R by hitting Ctrl-z. You will see a message that R has been stopped. Now you can type any other command, for example look for a certain data file, edit an R script, etc. When

[m] If you want to kill a job that is running in the batch queue (cf. Chapter 6), you can use the qdel command.

fg

you are ready to work in R again, type `fg`, which stands for “foreground”. Nothing seems to happen, but you are back in R. Hit the Enter key for example, or use the up arrow to redo one of your previous commands.

b) It is possible to have multiple programs suspended at the same time. Suspend R again and start `htop`. Suspend `htop` as well. Again a message appears telling you that `htop` has stopped. Notice that the number in between square brackets is now 2. This is the job ID (which is not related to the job ID of a job that runs in the SGE queue system that will be discussed in Chapter 6). You can list the jobs in your current screen with the `jobs` commands:

jobs

```
$ jobs
[1]- Stopped              /usr/bin/R --no-save
[2]+ Stopped              htop
```

You can check that both processes still exists by typing `ps -fu username`, like we did before.

Typing `fg` now would bring the last job to the foreground: `htop`. To bring R to the foreground instead type

```
$ fg 1
```

and exit R (type `q()`). Type `fg` to bring `htop` to the foreground and exit it (or, instead of bringing `htop` to the foreground kill it).

Nowadays the possibility of suspending jobs and bringing them back is not widely used anymore. You can simply open another connection to the server. In the old days, however, this was a completely different matter. Before the availability of Personal Computers (PCs) on each desktop, people connected to a server using a so-called terminal. Such a terminal consisted of a screen and a keyboard. Both were connected to a big mainframe computer somewhere down the hall. There were no mouses or windows (in the computer sense of those words), only one command line. In those days being able to suspend and run jobs in the background was vital.

Exercise E3.13 Starting processes directly in the background**

If you use MobaXterm it is possible to run certain GUI applications on the server. You could start Emacs for example. However, starting Emacs in the normal way:

```
$ emacs some_kind_of_file
```

will keep the shell occupied until you have finished editing your file. In this case you most likely want to start the process in the background so that you can do both: use the editor and use the command line. Starting a process in the background immediately can be achieved by ending the command with an ampersand symbol (which generally looks like & or &, depending in the font used):

```
$ emacs some_kind_of_file &
$
```

Notice how the shell prompt (the $) immediately returns to the screen, allowing you to run another command (while Emacs is still running in the background)[n].

3.12 Miscellaneous commands

In this section several small but useful commands will be discussed briefly. Use the man pages or the `--help` option to obtain more information on how to use these commands.

3.12.1 `wget`: downloading files to the server

Downloading files is an often occurring task. Of course you could download the file to your PC first and then use MobaXterm or WinSCP to transfer the file to the server. As you can easily imagine this is inefficient,

[n] This example assumes that you have X11 forwarding enabled when connecting to the server, see §3.1.1 for more information.

especially for large files. It's a waste of time and bandwidth. The solution is easy: look up the link of the file you want to download on your PC (right-click on the link in the web page and click and click 'Copy link location'). Next, use `wget` to download the file on the server. For example, to download the current version of ProbABEL[o] into your current directory run:

wget

```
$ wget http://www.genabel.org/sites/default/files/software/probabel-0.4.1.tar.gz
```

3.12.2 sort

The `sort` command is used to sort the lines in a file:

sort

```
$ more sort1.txt
1200
34
465
2340
Hello
Hello
1200
982
archive
1
Once upon a time
0001
$ sort sort1.txt
0001
1
1200
1200
2340
34
465
```

[o] Shameless plug: ProbABEL [6] is a tool for doing genome-wide association studies, and I am the current maintainer. See http://www.genabel.org/packages/probabel.

```
982
archive
Hello
Hello
Once upon a time
```

sort -n

Notice how `sort` treats numbers as if they are words (2340 comes before 34 because a 2 is less than a 3). The `-n` option does a numerical sort:

```
$ sort -n sort1.txt
archive
Hello
Hello
Once upon a time
0001
1
34
465
982
1200
1200
2340
```

sort -g

The numerical sort is not perfect, however, because it doesn't recognise numbers in scientific notation, like `1.4e-5`. The `-g` option solves that (see below for an example).

sort -k

In case you don't want to sort on the first column, use the `-k` option followed by the column number:

```
$ more sort3.txt
rs5 23 c
rs150 10 a
rs23 1e2 d
rs10 0.2 b
rs3 4e-3 e
$ sort -k3 sort3.txt
rs150 10 a
rs10 0.2 b
```

```
rs5 23 c
rs23 1e2 d
rs3 4e-3 e
$ sort -gk2 sort3.txt
rs3 4e-3 e
rs10 0.2 b
rs150 10 a
rs5 23 c
rs23 1e2 d
```

Note that we used the `-g` option in the last command in order to sort the second column using scientific notation. Here, the two options were combined, but the above is equivalent to `sort -g -k2 sort3.txt`.

sort -t

To specify a different column separator than a space, use the `-t` option[p]:

```
$ sort -g -k2 -t"," sort4.txt
rs3,4e-3,e
rs10,0.2,b
rs150,10,a
rs5,23,c
rs23,1e2,d
```

sort -u

Recently, a new option was added to the `sort` command. The `-u` option sorts first and then keeps all unique lines (i.e. it removes entries that occur twice)[q]. For example, when sorting the `sort1.txt` file, only one of the `Hello` and `1200` lines remains when using the `-u` option:

```
$ sort -u sort1.txt
0001
1
```

[p] When thinking about grouping multiple short options, remember that if you have more than one option that needs a argument, these cannot be combined. So, `sort -gk2t"," sort4.txt` is incorrect, but `sort -gk2 -t"," sort4.txt` or `sort -k2 -gt"," sort4.txt` are correct. However, in a case like this specifying all options separately is the most readable solution.

[q] Looking ahead at § 3.12.3 that deals with the `uniq` command and § 3.13.2 about using the pipe to send output from one command to another, the command `sort -u some_file` is identical to running `sort some_file | uniq`.

```
1200
2340
34
465
982
archive
Hello
Once upon a time
```

3.12.3 uniq

uniq

The `uniq` command removes repeated lines from a file.

```
$ uniq sort1.txt
1200
34
465
2340
Hello
1200
982
archive
1
Once upon a time
0001
```

Notice how only the repeated entry of “Hello” has been removed, but the duplicate “1200” still in because the entries are not on subsequent lines. In § 3.13.2 you will see how the `sort` and `uniq` commands can be chained together to also get rid of one of the “1200” entries.

3.12.4 `wc`: counting words and lines

Counting words and lines are common tasks. For example, counting the number of lines in a phenotype file will tell you if the number of individuals in the file corresponds to what you know to be the total number of wc

people in your study cohort. The wc (WordCount) command counts the number of lines, words and characters for each file that you give it as input. Here you see that the file sort1.txt has 12 lines, 15 words and 70 characters.

```
$ wc sort1.txt
12 15 70 sort1.txt
```

wc -l

In our field we are mostly interested in the total number of lines, in which case you can use the -l option.

3.12.5 date

date

date -d

To get today's date and time use the date command. To change the way the date and/or time are printed use + followed by a so-called format specifier. To use a different date than today, use the -d option. These are a few typical examples:

```
$ date
Thu Jun 4 12:36:45 CEST 2015
$ date +%F
2015-06-04
$ date +%H:%M
12:37
$ date +%H:%M-%Y-%m-%d
12:37-2015-06-04
$ date -d "15 November 2015 - 30 days"
Fri Oct 16 00:00:00 CEST 2015
$ date -d "15 November 2015 - 30 days" "+%d %B"
16 October
```

Notice the quotes around the + format specifier. They are necessary because of the space between %d and %B, without the quotes, only %d would be seen as part of the format specifier and %B would be considered a separate option (which doesn't exist, so an error will be returned). The output of the date command may be localised to your country. For example, on a Dutch system the results of the first and last examples are:

```
$ date
do jun 4 12:41:33 CEST 2015
$ date -d "15 November 2015 - 30 days" "+%d %B"
16 oktober
```

To override the language environment for a specific date command add `LANG=CC` in front of it, for example on the Dutch system the result will be English as expected:

```
$ LANG=CC date
Thu Jun 4 12:43:02 CEST 2015
$ LANG=CC date -d "15 November 2015 - 30 days" "+%d %B"
16 October
```

3.12.6 `du`: disk space usage

`du`

`du -sh`

If you are getting close to your disk quota limit, the `du` command (which stands for "disk usage" will help you find files and directories that take up a lot of space:

```
$ du -sh Projecten/
4.4G  Projecten/
$ du -sh Projecten/*
1000M Projecten/GWAS
18M Projecten/GvNL
2.7G  Projecten/ARA
104K  Projecten/ergo
16K Projecten/RS3_2082_ids
300K  Projecten/Comparison_polygenic_hglm
716M  Projecten/Lipid_GxE_Prediction
2.0M  Projecten/Suman_Prediction_Comparison
8.9M  Projecten/Epiblaster
```

The `-s` option summarises the disk usage for each of the directories, without it you will get a report for each subdirectory. The `-h` option works like the `-h` option of `ls`, it prints the size in human readable format (MB, GB, etc.) instead of in bytes.

3.12.7 Differences between files

diff

Finding the differences between two text files is easy with the `diff` command. `diff` shows each line where a difference occurs[r]. Consider the following two files, `file1` and `file2`:

```
$ more file1 file2
::::::::::::::
file1
::::::::::::::
Hello world,
I am a text file
living in a Linux world.

Bye bye
::::::::::::::
file2
::::::::::::::
Hello world,
I am a simple file
living in a Linux world.

Bye bye
```

This is the output of `diff`:

```
diff file1 file2
2c2
< I am a text file
---
> I am a simple file
```

diff -u

which tells us that there is a difference on line 2 in the first as well as in the second file and then prints the differing lines. The `-u` command shows the differences within the context by adding a few lines before and after the differing lines:

[r] Note that sometimes this may only be a difference in white space.

```
$ diff -u file1 file2
--- file1 2011-01-27 18:08:19.000000000 +0100
+++ file2 2011-01-27 18:08:19.000000000 +0100
@@ -1,5 +1,5 @@
 Hello world,
-I am a text file
+I am a simple file
 living in a Linux world.

 Bye bye
```

md5sum

For binary files it is more difficult to display the differences (do you understand why?) and diff will only tell you that the files differ. To check whether two files are equal bit by bit use the md5sum command which calculates a unique checksum based on the contents of the file. If the checksums are equal you can be assured that the files are exactly equal.

```
$ md5sum file1 file2
24adf0b0daed9b2b310c4e2117fcbdda file1
4b25f33b80a2514d524c3f5b60e13bd6 file2
$ cp file1 file4
$ md5sum file4
24adf0b0daed9b2b310c4e2117fcbdda file4
```

The checksums are different in the first run where file1 and file2 are compared. Making a copy of file1 and checking that file's md5sum shows that they are exactly equal.

md5sum -c

The md5sum command is very useful to check files before and after transfer to or from another server (cf. § 3.7) to make sure that all files are exactly equal. Checking a lot of files by hand in this manner is not efficient. The -c option helps here. It compares the md5sums of a set of files to those listed in a file. If the md5sums of a set of files is given (e.g. by saving the output of the md5sum command on the first machine) then after transfer the files can be checked using this file:

```
$ more checksums
24adf0b0daed9b2b310c4e2117fcbdda file1
```

```
4b25f33b80a2514d524c3f5b60e13bd6 file2
$ md5sum -c checksums
file1: OK
file2: OK
```

Exercise E3.14 Disk space usage

Find out which subdirectory of your home directory takes up most space.

Exercise E3.15 Downloading files to the server

When downloading large files, the transfer gets interrupted sometimes. If the happens, you could start downloading from scratch again, of course. But it's more efficient to continue downloading where you left of.

a) Start downloading the CD image of the most recent Ubuntu Linux Long Term Support version from this URL: `http://releases.ubuntu.com/16.04.3/ubuntu-16.04.3-desktop-amd64.iso`. Once the download starts, hit `Ctrl-c` to abort the transfer. Check the size of the .iso file with `ls`.

b) Which `wget` option allows you to continue downloading a partially-downloaded file? Try it.

3.13 Input and output redirection

3.13.1 Redirecting to and from files

When working on analyses that involve some repetitiveness (i.e. performing the same action for 22 chromosomes) it is easy to end up with several files that you would like to have in one big output file. Or maybe you want

>

>>

to store the screen output of a certain command because it might be useful later. The > and >> are used to send the screen output of a command to a file, this is called output redirection.

For example, the output of the `sort` command that sorted some input file `infile` normally goes to the screen. Sending it to an output file called `outfile` goes like this:

```
$ sort infile > outfile
```

The > and >> behave differently in one important way: > always overwrites the output file, whereas >> appends the new output to the output file if it exists. For example, you can send the listings of several directories to an output file `dirlist` (the `cat` commands simlpy print the output on the screen so you can follow what happens):

```
$ ls -l dir1 > dirlist
$ cat dirlist
total 8
-rw-r----- 1 lennart genepi 492 2011-10-16 15:42 file1
-rw-r----- 1 lennart genepi 150 2011-10-16 15:42 file2
$ ls -l dir2 >> dirlist
$ cat dirlist
total 8
-rw-r----- 1 lennart genepi 492 2011-10-16 15:42 file1
-rw-r----- 1 lennart genepi 150 2011-10-16 15:42 file2
total 8
-rw-r----- 1 lennart genepi 129 2011-10-16 15:42 file21
-rw-r----- 1 lennart genepi 14 2011-10-16 15:42 file22
```

<

Input redirection is complementary to output redirection, it is however, not used as much as its brother. With input redirection the contents of a file is sent to a program, which then processes it. The symbol used for input redirection is <. As an example, consider the following (useless) R script, saved in the file `rinput.R`:

```
print("Hello, you are now in R")
getwd()
1+1
10:1
```

One way to execute these lines in R is to use input redirection:

```
$ R --quiet < rinput.R
> print("Hello, you are now in R")
[1] "Hello, you are now in R"
> getwd()
[1] "/tmp"
> 1+1
[1] 2
> 10:1
 [1] 10 9 8 7 6 5 4 3 2 1
>
$
```

Notice that we have ended at the bash prompt again, even though we didn't end the script with `q()`.

3.13.2 Redirecting output of one command to another

So far we have sent the output of a command to a file, or taken the input from a file. Things start to get interesting when the output of one command is used as input to the next. For this the pipe symbol | is used. It usually on the same key as \ and indicated by a vertical bar or two vertical bars stacked like a colon).

You can use this for example when looking at the directory listing of a directory with many files and subdirectories. Instead of having the output scroll off the screen you can pipe it to e.g. `more`:

```
$ ls -l | more
```

Or if you want to want to look at all processes that are running on the system, screen by screen:

```
$ ps -ef | less
```

Technically using a pipe like

```
$ command1 | command2
```

is equal to

```
$ command1 > tmpfile
$ command2 < tmpfile
$ rm tmpfile
```

In later chapters, after more commands have been introduced, more examples of the usage of pipes will be shown.

Exercise E3.16 Combining files

In this exercise you will use output redirection to concatenate (i.e. stitch together) several files.

a) Create a new directory and `cd` into it. Use your editor to create three files with a few lines of text. Save them as `file1`, `file2` and `file3`.

b) Use the `cat` command and the > sign to send the contents of `file1` to a new file called `output.total`

c) How would you add the contents of the other two files to the `output.total` file?

d) How can you quickly check whether the files were merged correctly?

If the input files are large, don't forget to delete them after you have are satisfied with the result of the concatenated file. There is no reason to let this duplicate information eat up disk space from your quota.

Exercise E3.17** Combining input and output redirection

How would you send the output of the R commands from the input redirection example (`R --no-save --quiet < rinput.R`) to a file called `Routput`?

Exercise E3.18*** Using the output of one command as input for another

The pipe symbol (|) is used to send output from one command as input to another. In this way many short commands can be chained into a powerful ”supercommand”.

a) Chain two commands together to show how many processes you are running at the moment.

b) Go to your home directory and find out how much disk space (in human readable format) each of the files and subdirectories use. Sort the result in order of increasing size.

c) Use w to list all users that are currently logged in. How many users are currently logged in? Some people may be logged in more than once. How many unique users are logged in?

w

3.14 Aliases and creating your own commands

We are almost at the end of this chapter and already you have gone through many commands. How many times have you typed ls -l or ls -lhp? Probably quite a few times. And more will follow! Wouldn’t it be nice if you could save a command under a shorter name? Well, you can! The alias commands lets you assign a command to a new name. For example:

alias

```
$ alias lsl='ls -lhp'
```

Once you have run this command you can simply type lsl, three characters instead of seven. That saves time! Chapter 6 the qstat command will be discussed. This command accepts several options and typing qstat -f -u * many times is a pain, so why not write an alias and abbreviate it to qs?

```
$ alias qs='qstat -f -u \*'
```

You may have noticed if you have more than one terminal window open that aliases defined in one terminal session are not known in another. Moreover, if you close a terminal all aliases will be forgotten the next time you log in. Not a very nice thing, but, as in most cases, someone already ran into that problem and has solved it for you. The solution is to save your aliases in one of the two hidden files `~/.bashrc` or `~/.bash_aliases`. Notice the tilde (~) in the path, these two files reside in your home directory. The `.bashrc` file in your home directory is the default configuration file for Bash, it is read every time you start Bash (i.e. when you open a terminal). This file is already present on most Linux machines and you can simply add your aliases at the end of the file[s]. Many default `~/.bashrc` files also have a section that looks like

```
# Alias definitions.
# You may want to put all your additions into a separate file like
# ~/.bash_aliases, instead of adding them here directly.
# See /usr/share/doc/bash-doc/examples in the bash-doc package.

if [ -f ~/.bash_aliases ]; then
   . ~/.bash_aliases
fi
```

which basically means that every time you log in `~/.bashrc` checks whether the file `~/.bash_aliases` exists and if that is the case it reads its contents[t]. So, as the comment says "You may want to put all your additions into a separate file".

After you have added your aliases to either of these files the present Bash session doesn't know about them yet. You could log out and back in of course, but a more elegant way is to read the contents of the file directly:

[s] The `.bashrc` file is actually a Bash script, so everything you will learn in Chapter 5 can be applied to this file as well.

[t] Feel free to add these lines to your `.bashrc` file if they don't exist and you want to store your aliases in a separate file.

```
$ source ~/.bashrc
```

Do you remember from §3.6.2 that the -i option cp, mv and rm makes these commands safer by asking for confirmation before overwriting or deleting? If you'd like to be on the safe side this is the time to create aliases for these commands:

cp -i

mv -i

rm -i

```
alias cp='cp -i'
alias mv='mv -i'
alias rm='rm -i'
```

By giving the alias the name of the original command you "overwrite" the original command with your alias[u].

In exercise E3.18 of this chapter we saw how we could use du and sort to show the file and directories in the pwd sorted by file size, with the largest files and directories at the bottom. The full command was

```
$ du -sh * | sort -h
```

If you are wondering which files and directories eat most of your precious disk space quota you are probably interested in the top 5 largest ones in a given directory. Let's reverse the sort so the largest files and directories are at the top and then select only the top five of them:

```
$ du -sh * | sort -rh | head -5
```

It makes sense to save this nice string of commands in an alias so we don't have to remember it exactly:

```
alias bigfiles='du -sh * | sort -rh | head -5'
```

You can run the alias command all by itself to see which aliases have been defined. Here is a selection of my aliases:

```
$ alias
alias R='/usr/bin/R --no-save'
alias grep='grep --color=auto'
```

[u] Of course the real commands are not overwritten, but when you give a command Bash first looks in its list of aliases before looking in its set of default directories for executable files.

```
alias lo='exit'
alias ls='ls --color=auto'
alias lsa='ls -lhA'
alias lsl='ls -lh'
alias qs='qstat -f -u \*'
```

Exercise E3.19 Creating aliases

After reading/hearing all the warnings about `cp`, `mv` and `rm` overwriting an existing destination file, you may want to decide to stop living on the edge and use more safe alternatives for these commands.

a) If you haven't already read about them in the course reader, find out which option to add to the `cp`, `mv` and `rm` commands that will force those commands to ask you a question before performing a (potentially) dangerous operation

b) Use the `alias` command to "overwrite" each of these three existing command with its safer alternative and test the alias.

c) Open the file `~/.bashrc` in an editor and add the three aliases you created earlier.

4

Chapter 4

Working with text files

Data files fall into two categories, files that store information in plain text and those that store information in a binary format. Binary formats are in general more space-efficient than text files, but have the drawback that they can't be opened with a regular text editor (cf. § 3.2) and can usually only be read and written by the programs that created them. Examples of binary files are `.Rdata` files, zip files and GenomeStudio's project files, but also the `.doc` and `.xls` files from MS Office. Plain text files, sometimes also called ASCII files, are much easier to handle. As long as they are small they can be edited easily, without the need of special programs. However, as they grow in size, up to several GB for a file with imputed data for chromosome 1 in ERF, things become problematic. For example, to transfer a file of 10 GB from the server to your office PC via the 100 Mb/s office network would take at least

$$\frac{10 \times 1024\,\text{MB}}{100\,\text{Mbit/s}} \times 8\,\text{bits/byte} = 819.2\,\text{s} \approx 14\,\text{min}$$

and then you would still need to do the editing, and you could run into trouble there to, because the file is larger than the roughly 4 GB of memory (RAM) installed in your PC. And finally, you would need to upload the file again after finishing the edit.

Even if you would try to edit the same file directly on the server using one of the editors mentioned in § 3.2 (which would, of course be the smarter thing to do), the loading time would be considerable. That is why in this chapter you will learn to use several utilities for processing (large) text files. Most of these utilities parse the files line by line, which solves the memory problem described earlier.

4.1 Converting between Windows and Linux format

Some Linux programs may have difficulty reading plain text files created with a text editor on Windows (e.g. Notepad). This is caused by the fact that historically both operating systems use a different way to encode the end-of-line. For example, opening a text file that was created in Linux on a Windows PC (with Notepad, for example) will show all text on one long line. Conversely, a Windows file opened with e.g. `less` will show the Windows end-of-lines as `^M`. Two programs exist to solve this problem: `dos2unix` and, for the reverse conversion, `unix2dos`[a].

`dos2unix`

`unix2dos`

4.1.1 Exercises

Exercise E4.1 Converting files from Windows format to Linux format

a) By default `dos2unix` does the conversion in place, i.e. the original file is overwritten. Which command line option should be given to write the converted output to a new file?

It is not possible to see the difference between a file created in the Windows format and one in the Linux format when using paging tools like `more` or `less`. Most Linux systems will display and use the Windows files without problems.

The Linux command `file` tries to classify each file you give as an argument. For a `.jpg` image, for example it will give the following results:

`file`

```
$ file linux-penguin.jpg
linux-penguin.jpg: JPEG image data, JFIF standard 1.0
```

This is useful, because if I would make a simple text file and name it `text.jpg`, having the `.jpg` extension alone does not automatically make it an image. And the `file` command tells me so:

```
$ cat text.jpg
Hello there, this is a text file
$ file text.jpg
text.jpg: ASCII text
```

[a] For those of you who are young: MS-DOS was the first OS created by Microsoft, back in 1981, way before they created the MS Windows family.

b) Create a text file with your favourite editor. Use the `file` command to check its type. Convert it to Windows/DOS format without overwriting the old one and check the type of the new file.

4.2 `grep`: finding text

grep

The `grep` command is used to quickly find lines containing a given text pattern in one or more files. The basic syntax is

```
$ grep searchpattern files
```

If `searchpattern` consists of more than one word it has to be enclosed in quotes.

The most-used options for `grep` are:

grep -F

- `-F` treat the search text as fixed, litteral text, not as a pattern. By default the search text is a so-called basic regular expression, not just litteral text (cf. § 8.1). This speeds up the `grep` process enormously, something very useful given the enormous size of many of the files used in the life sciences. So, unless you know what regular expressions are and how to use them I strongly encourage you to always add this option!

grep -w

- `-w` assume that the search text is a word (and thus should be surrounded by white space (spaces, tabs)). Can be very important! For example, if you are looking for a person with ID `id10`, using `grep id10 myfile.txt` will also print lines containing `id100`, `id10a`, `fid10212`, etc. The `-w` option will only return the lines you are actually looking for.

grep -i
grep -n
grep -c

- `-i` makes the search case-insensitive
- `-n` adds the line number on which the match occured to the output
- `-c` (count) don't print the lines matching the search text, but only print the number of lines with a match

grep -v
grep -A
grep -B

- `-v` print all lines not matching the search text
- `-An` Print `n` lines after the match
- `-Bn` Print `n` lines before the match

p -C

`-Cn` Print `n` lines before and after (the `C` stands for “context”) the match

Let’s have a look at some examples with a slightly modified version of the file used for sorting in § 3.12. First, print the contents of the file on the screen:

```
$ cat grepexample.txt
1200 1234 4567
34 some text here
here is a number: 465
2340
Hello
Hello there
1200
982 id120 exm12312
the archive is lost
1
Once upon a time
0001
```

Now, print only the lines containing the text `34`.

```
$ grep 34 grepexample.txt
1200 1234 4567
34 some text here
2340
```

Print the lines containing a `34` “on its own”, i.e. surrounded by white space:

```
$ grep -w 34 grepexample.txt
34 some text here
```

Notice the difference between the last two outputs.

The `-n` option adds the line number on which the match was found:

```
$ grep -n 0 grepexample.txt
1:1200 1234 4567
4:2340
7:1200
```

```
8:982 id120 exm12312
12:0001
```

If you just want to count how many times a certain text occurs, use the `-c` option:

```
$ grep -c 0 grepexample.txt
5
```

The following examples show how to use the after, before and context options:

```
$ grep -A2 archive grepexample.txt
the archive is lost
1
Once upon a time
$ grep -B2 archive grepexample.txt
1200
982 id120 exm12312
the archive is lost
$ grep -C1 archive grepexample.txt
1200
982 id120 exm12312
the archive is lost
1
$ grep -A2 1 grepexample.txt
1200 1234 4567
34 some text here
here is a number: 465
--
1200
982 id120 exm12312
the archive is lost
1
Once upon a time
0001
```

In the last example the `--` are inserted to separate the output for the first match (the line with `1200 1234 4567`) and the second match (the line with only 1200). There is no -- between the second and the third, fourth and fifth matches (the lines with id120, 1 and 0001, respectively), because these overlap or are directly connected. , and

grep -f

If you have more than one search term it is easiest to save them in a file, one term per line, and feed that file to grep with the -f option. For example, if you have a list of SNPs (e.g. their rs IDs) in a file called snplist you can look up the information about these SNPs from the genotype imputation files with

```
$ grep -Fwf snplist /path/to/your/imputation/*.info
```

I've added the options -F and -w as well to speed up the search and only look for matches of the full word.

4.2.1 Exercises

Exercise E4.2 Searching for a given text in a file

a) Given a phenotype file, how would you check whether a certain individual ID is present?

b) Go to the directory with the extracted exercise files (cf. Exercise E3.10 on page 37).

c) Which grep option allows you to recursively search in directories? Use it to find all occurrences of the term "trait" in the subdirectories.

d) In the output of the previous command there is also a line that has the word "traits" (plural) in it. If you are only looking for the single word "trait", which grep option would you use?

e) By default, grep is case sensitive. Find out how to do a case insensitive search and see whether there are occurrences of the word "trait" with capital letters in them.

f) Spotting the exact location of the keyword in the output of `grep` is sometimes difficult. Try the `--color=auto` option (or `--colour=auto` if you are British) for a few of the previous `grep` commands and see the difference.

4.3 `sed`, the Stream EDitor

sed

An operation that occurs frequently is replacing a certain text in a file with other text. The command `sed`, for Stream EDitor, can do this (and a lot of other things) very well. By default `sed` sends its output to the screen so you have to used output redirection (cf. § 3.13) to send it to a file. The structure of a `sed` search-replace operation looks like this:

```
$ sed 's/old text/new text/g' some_file
```

Here the `s` stands for "substitute" and the `g` for "global", without `g` only the first occurence of `old text` in each line is replaced. Sometimes you need to substitute the / character itself (if you are changing a directory path for example). This can be awkward with the normal `sed` substitution command. Luckily the / symbol in the `sed` command can be replaced with other symbols, for example a colon or semi-colon. The following command is equal to the one above.

```
sed 's;old text;new text;g' some_file
```

By default `sed` sends its output to the screen. For this small file that is not a problem, but for a large file your screen will soon be too small. It would be much nicer to have this output in a separate file. There are two ways to do this. The first one is obvious after the exercises in § 3.13: use > to send the output to a new file. The second method will be discussed in the exercises.

Another common use of `sed` is to remove a given line, say line 3, from a file:

```
$ more some_file
This is a header
Line 2
Line 3
Line 4
Line 5
Line 6
This is the footer
$ sed '3d' some_file
This is a header
Line 2
Line 4
Line 5
Line 6
This is the footer
```

In the next two examples we first delete lines 2 to 4 and then show how to delete all lines containing the text “This is”.

```
$ sed '2,4d' some_file
This is a header
Line 5
Line 6
This is the footer
$ sed '/This is/d' some_file
Line 2
Line 3
Line 4
Line 5
Line 6
```

To print only line 5 of a file use

```
$ sed -n '5p' some_file
Line 5
```

Without the `-n` option `sed` prints all lines in the file.

4.3.1 Exercises

Exercise E4.3** Using sed for search-replace operations

a) Go to the directory that contains the exercise files you extracted from the tar.gz file in Exercise E3.10. Go to the directory called Exercise_sed. List the contents of the directory.

b) There should be one file called file.csv, a file with comma-separated values. Show the contents of the file.

c) Some programs only accept tab-delimited files as input, so we have to replace all comma's with tabs. Tabs are indicated by \t in many programs. Write a sed command to replace the commas with tabs.

d) As mentioned earlier, sed sends its output to the screen by default. Send the output to a file called file.tsv using output redirection.

sed -i

The second method is to edit the file "in place" instead of creating a new file. This is especially handy for large files. The sed option -i is used to specify in-place editing. Be careful, there is no way back!

e) Write a sed one-liner that replaces all tabs in the .tsv file with semi-colons. Use the -i option.

4.4 cut: selecting columns

A task you will come across often is selecting columns from a data file for further processing. For example, one of the output files of genotype imputation programs is the so-called info file. This file contains information on the imputed SNPs (one SNP per row), with the first column being the rs ID or chromosome:position combination, the second and third columns being the major and the minor allele, the fourth column being the MAF, etc. The 7th column usually contains the R^2 values, which are a measure for the imputation quality.

`cut`

To select for example only the SNP ID and the R^2 value you can use the `cut` command. A typical cut command looks like this:

```
$ cut options file
```

`cut -f`

The most important command line option for `cut` is `-f`. The `f` stands for "field" (a.k.a. column). This option is followed by one or more numbers that tell `cut` which columns to print, for example

```
$ cut -f 1 file
$ cut -f 1,2,6 file
$ cut -f 6-10 file
$ cut -f 4- file
```

The first line prints the first field from `file`, the second example prints columns one, two and six. The thrid example prints columns 6 to ten, and the last example prints columns 4 and higher.

`cut -d`

By default `cut` assumes that the columns are separated by a TAB, if that is not the case in your file you have to specify the delimiter on the command line using the `-d` option followed by the delimiter in double quotes. For example, to tell `cut` that columns are separated by a space use `cut -d " "`, if the values are separeted by comma's (a `.csv` file), use `cut -d ","`, etc.

4.5 GAWK: more fun with columns

GAWK is like a big brother of `cut`. It is a scripting language that is mostly used with data that can be divided into records and fields. This may sound a bit abstract but if you replace "records" with "lines" and "fields" with "columns" (just like we did with `cut`) you'll get the idea. In fact, you can tell GAWK what to consider as a record and what as a field by telling it which characters to use as record separators and which to use as field separators. By default fields are separated by white space (spaces, tabs) and records are separated by a newline characters.

The name GAWK stands for GNU AWK. AWK is the original program from before the 1980s and is named after its original authors: Al Aho, Peter

J. Weinberger and Brian Kernighan. GAWK extends the AWK language, but for most common tasks the languages are the same. On modern Linux systems the awk program is usually a link to the gawk program.

gawk

The basic structure of GAWK commands is as follows:

```
$ gawk 'condition {action}' file
```

For each line (record) in the file GAWK will test the condition and if it is true it will perform the action, usually modifying or printing a column (field). Like so many other tools, gawk sends its output to the screen, so use output redirection (§ 3.13) to send the output to a file.

Fields in GAWK are noted by $1, $2, etc. (for the first and second field). The last field is denoted by $NF and $0 denotes the whole record (line). Let's illustrate this with a couple of examples. In the sed exercises you created a file called file.tsv with tab-separated columns, let's print columns 2 and 4:

```
$ gawk '{print $2, $4}' file.tsv
field1 field3
12 14
22 24
```

There are a few things to note about this example. First, the "condition" is missing in the gawk command. This is no problem, it simply means that it should perform the "action" for every line. Second, the "action" is print $2, $4 which means print columns two and four, notice the comma, it is necessary (check what happens if you forget it)! The third point to note is that the first line of the output seems to be wrong, you would expect field2 field4, wouldn't you? However, check the first line of the input file:

```
$ head -n 1 file.tsv
# field1 field2 field3 field4
```

gawk -F

By default gawk uses all forms of whitespace as field separator (and not just tabs as you might have wanted), so $1 is the # and $2 is field1 and therefore line 1 has a total of five fields. If you want gawk to use a different field separator you can specify it with the -F option:

```
$ gawk -F "\t" '{print $2, $4}' file.tsv
field2 field4
12 14
22 24
```

In the following example the "condition" will be used to only print lines where the second column contains a 2:

```
$ gawk -F "\t" '$2 ~ "2" {print $0}' file.tsv
11 12 13 14
21 22 23 24
```

If you want to be more strict and only print lines that have exactly a 2 in the second column use `==` instead of `~`:

```
$ gawk -F "\t" '$2 == "2" {print $0}' file.tsv
```

You see: no result. If it doesn't matter in which column the 2 appears use

```
$ gawk -F "\t" '/2/ {print $0}' file.tsv
# field1    field2  field3  field4
11  12  13  14
21  22  23  24
```

Of course this shows all lines in this case. Note that since we print the whole line it would have been easier to use `grep` here.

If more than one condition needs to be satisfied before we want a line to be printed, this can be done as well, for example by using the logical operators "and" (`&&`) or "or" (`||`). For example, the following prints only the lines where the value in the second column is larger than 10 and the last column is less than 20:

```
$ gawk -F "\t" '$2 > 10 && $4 < 20 {print $0}' file.tsv
11  12  13  14
```

GAWK can not only be used to extract text, but also to change it. The first example below prints all columns except the second and third. It does so by changing the value of these fields to an empty string (`""`) before printing the whole line. The example after that uses the same technique to change the third column to NA:

```
$ gawk '{$2=$3=""; print $0}' file.tsv
#  field3 field4
11  14
21  24
$ gawk '{$3="NA"; print $0}' file.tsv
# field1 NA field3 field4
11 12 NA 14
21 22 NA 24
```

Notice that in both cases the header is also changed, which is probably OK in the first example, but not in the second. To make GAWK ignore the header we have to make use of the fact that GAWK partitions its input into records and fields (by default lines and columns as discussed before). To make GAWK leave the header untouched we check the value of NR, which always contains the current record number (NR is somewhat similar to NF which always contains the total number of fields in the current record):

```
$ gawk '{if (NR!=1) {$3="NA"}; print $0}' file.tsv
# field1 field2 field3 field4
11 12 NA 14
21 22 NA 24
```

The explanation of this command is: “For each line in file.tsv do the following: If the record number (the line number in this case) is not equal to one, set the third column to NA. Then (for all records) print the whole line”. In a similar way a combination of record number and field number can be used to print one element of the file, say the second column of the third line:

```
$ gawk '{if (NR==3) print $2}' file.tsv
22
```

4.5.1 Exercises

Exercise E4.4*** Creating a phenotype file from .ped data

In this exercise you will learn how to extract data from a file and reformat it for use by another program. Tasks like this are very common because most programs restrict themselves to one particular task (the UNIX philosophy for programs is “Write programs that do one thing and do it well” [7, Chapter 1.6], which is why you have to learn about so many small programs in this course).

a) Go to the directory that contains the exercise files you extracted from the tar.gz file in Exercise E3.10 in § 3.9.4. Go to the directory called Exercise_ped2phe.

b) Look for the file chr.ped.

The first six columns of this file in Merlin[b] .ped format are FAMILY_ID, PERSON_ID, FATHER_ID, MOTHER_ID, SEX and AFFECTED, the other columns contain genotypes.

c) Show the first six columns of the first ten lines of the .ped file.

d) In your answer to the previous question you have probably made use of the head command and the | symbol. Did you put head in front of the pipe symbol or after (as in some command | head vs. head chr.ped | some command)? What’s the difference?

A simple .phe file as used by e.g. ProbABEL [6] consists of the following header and the corresponding column data: id (the PERSON_ID from the .ped file), sex, bt1; the last column is for a binary trait, in our case the AFFECTED status.

e) Create a .phe file (without the header[c]) from the complete .ped file (not only the first lines). Hint: use output redirection to create the new file.

[b] See http://www.sph.umich.edu/csg/abecasis/merlin/tour/input_files.html for more info on the file format.

[c] It is possible to add the header in the same one liner as well, cf. the examples in the GAWK manual at http://www.gnu.org/software/gawk/manual/gawk.html#Print-Examples.

One problem remains, however. In `.ped` files women are coded as 2, men as 1. ProbABEL expects 0 for women. GAWK can be used for more than just printing columns, it is a script language of its own. In GAWK several commands can be written on a single line by separating them with a semicolon. Variables exist as well, as do if-clauses and for-loops. The one-liner that fixes both the header and the sex is shown in the answers.

4.6 Putting it all together

4.6.1 Exercises

Exercise E4.5**** **Filtering output using `gawk` (*thanks to Najaf Amin*)**

In this exercise GAWK will be used to filter output of another program in such a way that we end up with only the parts we are interested in.

So far GAWK has been used in so-called one-liners, relatively simple commands that only do one or two things. For the task at hand we will take a look at a GAWK script to get a glimpse of the full power of the GAWK language.

a) If you haven't downloaded and extracted the `tar.gz` file with exercise data yet (Exercise E3.10, page 37), do so now. The files for this exercise can be found in the directory `Exercise_gawk_snps`. Go to that directory and see which files are there. What is the file size of the files?

The `.awk` file is the GAWK script. The file `screen.1.out` contains the output of some quantitative trait analysis.

b) Use a pager to browse through the contents of the output file. After a header of several lines the data is quite regular and consists of tables for different SNPs for a number of traits.

GAWK is great when working with fields as has been shown in several of the previous exercises. Fields are denoted by $1, $2, $3, etc.[d] The last field is indicated by `$NF`, where `NF` stands for “Number of Fields”. Normally, fields are separated by white space but this can be changed. In order to check whether a certain field contains a certain `word` use the ~ operator. For example:

```
$2 ~ /word/ {print $0}
```

c) Compare the differences between the following GAWK one-liners and explain.

```
$ gawk '/trait/ {print $0}' screen.1.out
$ gawk '$2 ~ /trait/ {print $0}' screen.1.out
```

Back to the task at hand: writing a filter script for the output. Najaf wanted to extract from this output only those SNPs that had a *p*-value listed for allele 1. For each of those SNPs she wanted the both the *F* and the *p* columns printed. The names of the traits should be listed as well. A typical part of the `screen.1.out` file is reproduced here:

```
Testing trait:                NRUWE
=============================================

Testing marker:             rs884080
---------------------------------------------

 Allele df(0) Rsq(0) df(I) Rsq(I)   F     p
     1   366   0.05   365   0.06  3.13 0.0776 ( 241/372 probands)
     2   366   0.05   365   0.06  3.13 0.0776 ( 241/372 probands)

Testing marker:            rs2017143
---------------------------------------------

 Allele df(0) Rsq(0) df(I) Rsq(I)   F     p
     1   408   0.04   407   0.04  0.62      ( 278/414 probands)
     2   408   0.04   407   0.04  0.62      ( 278/414 probands)
```

The desired filtered output for this part should be

[d] Note the difference with Bash (Chapter 5) where these would refer to the command line arguments.

```
trait: NRUWE
rs884080 3.13 0.0776
```

d) Before starting to write a script it is always good to describe the steps that need to be taken in a schematic way. Without thinking in any programming or scripting language code, can you write down which steps should be taken to generate such output?

This is the contents of the `get_sign_snps.awk` script (lines starting with # are comments):

```
1 # This is a GAWK script that summarises
2 # the output of QT analysis
3 $2 ~ /trait/ {print $2, $NF};
4 $2 ~ /marker/ {snp = $NF};
5 $1 ~ "1" { if ($7 != "(") {
6       print snp, $6, $7;
7    }
```

Let's walk through it line by line. Lines one and two contain comments. Like in R[e)] any line that starts with a # is treated as a comment and is ignored by GAWK. On line three the first serious GAWK command is listed: "If field 2 contains the word *trait*, print both the second and the last field". In the next line we look for the word *marker* in the second field. If that is the case, we store the contents of the last field (the name of the marker) in the variable `snp` so that it can be used later on. Lines 5 – 8 are a bit more complicated because the command that is executed if the pattern is matched is not a simple print statement, but an if-clause. What these lines say in plain English is this: "If field one contains the value 1, take a look at field 7. If field 7 is not equal to the opening bracket ((that means the *p*-value column is not empty), print the variable `snp` we have saved in line 4, followed by fields 6 and 7 (containing the *F* and *p*-value)." Compare this with the schema you have made previously.

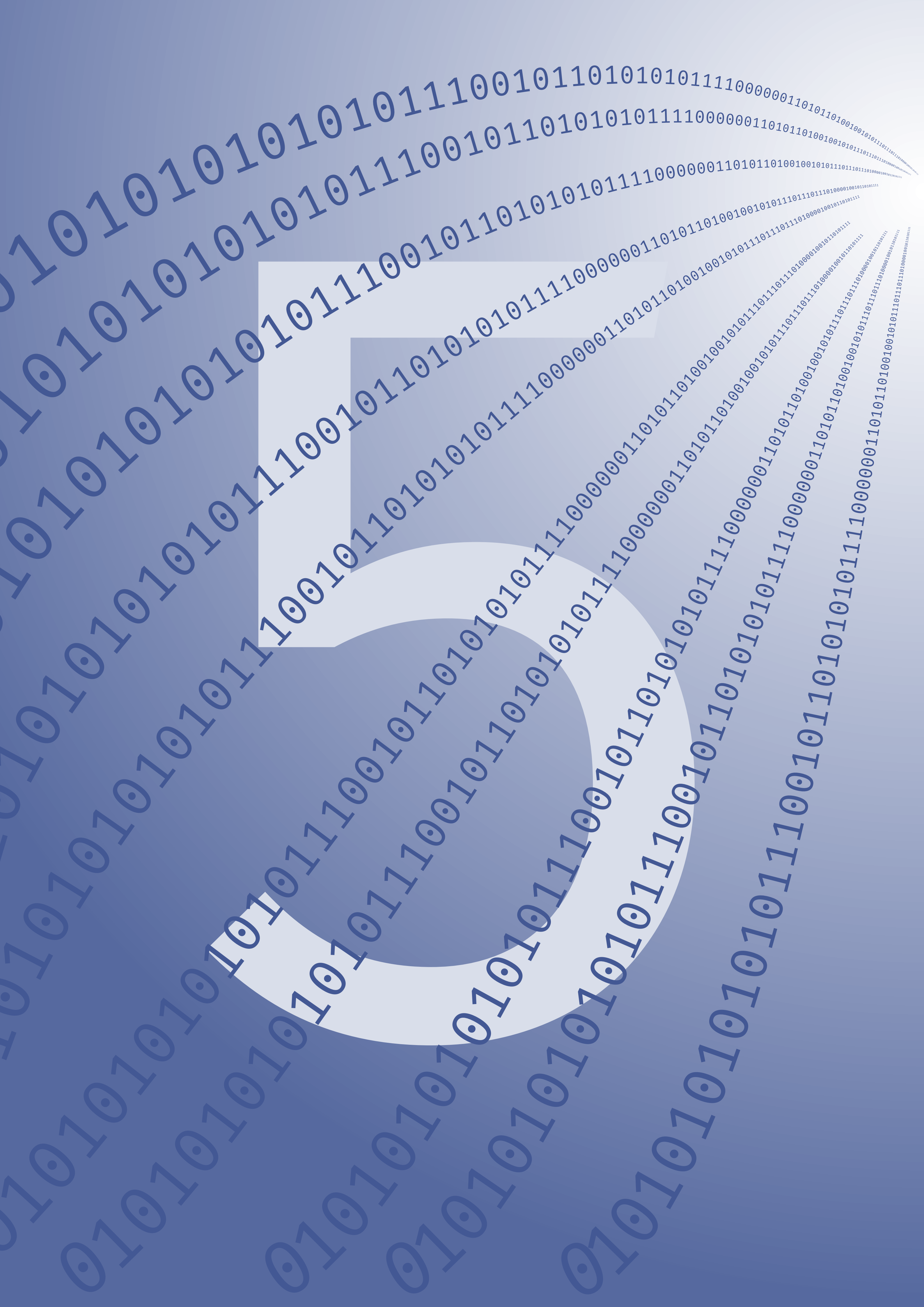
e) Which command line option is needed to run a GAWK script instead of the "normal" way where the GAWK commands are read

[e)] As well as several other scripting languages like Bash (Chapter 5) and Perl.

from the command line? Run the GAWK script and send the output to a file.

f) How many lines does your output file have?

g) How many traits were used for this analysis? And what where their names?


5

Chapter 5

Writing Bash scripts

On most GNU/Linux systems the default shell is Bash. Besides its 'normal' tasks as a shell (e.g. presenting the command line prompt to you, keeping track of your foreground and background jobs as well as your command history) it also allows you to write scripts. Scripts are an easy way to automate repetitive or complex tasks. Any command you type on the command line can appear in a script and *vice versa*, any command you see in a script should work on the command line (although you have to take care when using variables, making sure they have been defined, etc.).

5.1 A simple script

The standard ingredient of every Bash script is that it starts with the following line:

```
#! /bin/bash
```

The line helps the shell to understand in what kind of language the script is written[a]. A second ingredient in any script is the use of comments that explain what the script does. In Bash comments are lines starting with a hash (#). Writing output to the screen is a basic requirement of every script in any language. Bash uses the echo command for that:

echo

```
echo "This is my super script"
```

Lets make a small script with the information we have so far and call it first_script.sh. It is custom to give shell scripts the .sh extension, but unlike in Windows that does not mean that every file that has a name ending in .sh will be run by Bash.

Here is the contents of a simple script:

```
$ cat first_script.sh
#! /bin/bash

# This is my first script that prints text on the screen
```

[a] Other languages like Perl have a similar first line.

```
echo "This is my super script"
```

You can run a Bash script in several ways. The first one is perhaps the most straight forward:

```
$ bash first_script.sh
This is my super script
```

The second way is used more often, it involves making the script executable and then running it:

```
1 $ ls -l first_script.sh
2 -rw-r----- 1 lennart lennart 101 2011-10-25 09:24 first_script.sh
3 $ chmod a+x first_script.sh
4 $ ls -l first_script.sh
5 -rwxr-x--x 1 lennart lennart 101 2011-10-25 09:24 first_script.sh
6 $ ./first_script.sh
7 This is my super script
```

In line 2 the permissions of the script are shown. The execute permission (cf. § 3.10) is not set, so in line 3 we set it. As a result we can run the script as shown in line 6. Note the `./` in front of the file name of the script. As you remember from § 3.6.1, the `.` shortcut (when talking about directories) means the pwd, so here we explicitely state that we want to run the script `first_script.sh` located in the present directory. Whithout an explicit statement of the location of the script it won't run. You can try to run

```
$ first_script.sh
first_script.sh: command not found
```

but as you can see, that doesn't work[b]. The reason for this is that (unless you specify the path to an executable script of program explicitly) the shell only looks for executables in a few pre-configured directories and...the pwd (i.e. `.`) is not one of those directories. As a result you get the `command not found` message[c].

[b] As a matter of fact, you might have noticed that Tab-completion also didn't work and you had to type the full name of the script by hand.

[c] The reason for not looking for executable programs in the pwd is a safety measure. Imagine that you have an empty file called `ls` in the pwd. Had the pwd been in the

bash -x

As your script grows the probability that you make typo or some other error also increases. To check where in the script the error occurs use the -x option to bash. Take a look at the following (useless) script and try to spot the mistake.

```bash
#! /bin/bash
# A script to show how 'bash -x' can help finding errors.

echo "Welcome to the test script"

time=5
echo "This script will wait two times for ${time} seconds"
sleep ${time}
sleep ${tine}
echo "Done"
```

Running this script in the normal way gives an error:

```
$ ./errortest.sh
Welcome to the test script
This script will wait two times for 5 seconds
sleep: missing operand
Try `sleep --help' for more information.
Done
```

From this it is already easy to see that the problem lies with one of the sleep commands. Running it with `bash -x` will point out which one:

```
$ bash -x ./errortest.sh
+ echo 'Welcome to the test script'
Welcome to the test script
+ time=5
+ echo 'This script will wait for 5 seconds'
This script will wait for 5 seconds
+ sleep 5
+ sleep
```

search path for executable files this empty `ls` would be found and running `ls` would not give any results at all in that directory, while `ls` would have worked as expected in all other directories. To find out the default search paths type: `echo $PATH`.

```
sleep: missing operand
Try `sleep --help' for more information.
+ echo Done
Done
```

The lines starting with + are the lines as they appear in the script, but with all variables filled in. This output shows that the problem lies with the second sleep command. Like the first one there should have been a 5 following it. Closer inspection shows that there is a typo in the variable name. This process of finding a bug in a script is called “debugging”.

Exercise E5.1 A simple script

In this exercise you are going to write a script similar to the one in the text to get used to the Bash language.

a) Go to ~/LinuxCourse/ and open a new file in your favourite editor.

b) Now use the echo command to print the text Hello World on the screen. Save the file (in order to point out that the file is a shell script, it is customary to use the extension .sh for a script file name.)

c) In order to be able to run the script it should be made executable by setting the executable bit. Find out what the permissions (read, write and or execute) are set on your script file.

d) Set the executable bit with the chmod command and run the script by typing its name preceded by ./, e.g.

```
$ ./myscript.sh
```

5.2 Using variables

With the knowledge of the previous section simple scripts can be written. For example, you can write a Bash script that sets up a directory for a new project by copying files and subdirectories from a template set and then run a series of pre-tested commands, filter the output and finally remove any intermediate results. One of the major things missing, however, is how to use variables. That will be the topic of this section.

The use of variables in a script is very important. Variables allow you to make general, versatile scripts as well as saving you a lot of typing. Moreover, the use of clear variable names can greatly help someone else (or yourself, six months from now) to understand what the script does (or is supposed to do).

In bash variables are defined in the following way:

```
var="Some text"
```

It is important to note that spaces are not allowed around the = sign. To use a variable add a $ and (as a safety precaution) enclose the variable name in curly braces ({ }), like this:

```
echo "The contents of variable var is: ${var}."
```

Putting these steps in a complete script would look like this:

```
#! /bin/bash

var="I am a variable"

echo "The contents of variable var is: ${var}."
```

To put the output of a command into a variable use the $( ) construction. This is how to put the number of lines of a certain file into a variable, for example:

```
linecount=$(wc -l myfile.txt | gawk '{print $1}')
```

The pipe to GAWK is necessary because the output of `wc -l` not only prints the number of lines, but also the file name, in which we are not interested. Here is a complete script:

```
$ cat countMyLines.sh
#! /bin/bash
# This script counts the number of lines it contains

```

```
linecount=$(wc -l countMyLines.sh | gawk '{print $1}')

echo "This script contained ${linecount} lines."
$ ./countMyLines.sh
This script contained 6 lines.
```

This script can be enhanced a bit, because as it is now the file name of the script is hard coded, i.e. if the script is renamed, we need to change the code in the script as well to make it work as expected. In Bash scripts the variables `${1}`, `${2}`, etc. contain the arguments given on the command line when calling the script and `${0}` contains the name of the script itself. Note that these variables have nothing to do with the variables for fields/columns in GAWK, even though they follow a similar naming scheme. Using this information in the script we get:

```
#! /bin/bash
# This script counts the number of lines it contains

echo "The name of this script is ${0}."
echo "The first two command line arguments are ${1} and ${2}."

linecount=$(wc -l ${0} | gawk '{print $1}')

echo "This script contained ${linecount} lines."
```

Running the script we get

```
$ ./countMyLines2.sh
The name of this script is ./countMyLines2.sh.
The first two command line arguments are is and .
This script contained 9 lines.
$ ./countMyLines2.sh arg1 arg2
The name of this script is ./countMyLines2.sh.
The first two command line arguments are is arg1 and arg2.
This script contained 9 lines.
```

The first time the script is called without arguments. As a result nothing is filled in in the printed line. The second time the script is run two arguments are given and the are printed as expected.

As shown above, when printing text with `echo`, characters like `$` and `{}`, but also `!`, get interpreted by the shell and as a result are not printed on the screen. To print text literally use single quotes instead of double quotes:

```
$ echo "Hello, ${}, #, !"
bash: Hello, ${}, #, !: bad substitution
$ echo 'Hello, ${}, #, !'
Hello, ${}, #, !
```

Exercise E5.2** Using variables

Add a variable to the script of the previous exercise. The variable name should be `greeting`, because we would like the script to print `Good` ↩ `morning World` if the variable has the value `Good morning`.

Exercise E5.3** Using command line arguments in your script

Scripts become much more useful when they accept command line argument. You could, for example, make a script that runs a certain analysis for one chromosome. But then you would have to modify the script each time you want to run it for another chromosome. By using command line arguments the script can be run for any chromosome you want, for example, for chromosome 16 you would run:

```
$ ./myscript.sh 16
```

a) In a shell script the command line arguments are automatically saved in the variables `${1}`, `${2}`, etc. Write a script that print the first three argument in reverse order.

b) Modify the script from exercise E5.2 in such a way that instead of the ”Good morning” greeting it uses the first argument as greeting.

c) Arguments are normally separated by spaces, you might have come across this in the previous question. Can you think of a way to work around this problem?

5.3 Using shell variables in GAWK

Since a Bash script is nothing more than a set of commands that can also be run on the CLI, it is quite common to use other scripting languages within a Bash script.

This section shows how to send the contents of a Bash variable to GAWK. One reason to do this would be to make use of GAWK's column filtering capabilities, another would be the fact that doing arithmetic in Bash is difficult, whereas it is easy to do in GAWK. Consider for example a script that parses a set of files for a given SNP rs name and in one of the steps we would like to print only a set of given columns from a ProbABEL GWAS output. A schematic script might look like this:

```
#! /bin/bash
# This script does all kinds of stuff while looking for SNPs

# The first argument for this script is a SNP rs name
rsname=${1}
# The second argument for this script is a ProbABEL file name
filename=${2}

# Do other things here...

# Print only columns MAF, Rsq and loglik for the given SNP
gawk '$1==snpname {print $5, $7, $NF}' ${filename}
```

The question here is how to get the contents of the Bash variable `rsname` into the GAWK variable named `snpname` in the last line. Because the dollar signs mean different things in Bash and in GAWK some form of "translation" is necessary. This can be done using the `-v` option of GAWK. A simple example is to simply print the value of the variable on each of the lines of the output:

gawk -v

```
$ gawk -v var="some text" '{print var, $1, $3}' file.tsv
some text # field2
some text 11 13
some text 21 23
```

Here we used the tsv file created in one of the exercises on sed in § 4.3. Alternatively, to print the third column of a file we could use the following:

```
$ gawk -v col=3 '{print $col}' file.tsv
field2
13
23
```

Going back to the problem in line 12 of the example script above we can now see that this line should read:

```
gawk -v snpname=${rsname} '$1==snpname {print $5, $7, $NF}' ${filename}
```

What is done here is that the contents of the shell variable S{rsname} is copied to the GAWK variable snpname[d]. In the "regular" part of the GAWK command we check whether column 1 is equal to snpname and if so, we print column 5, 7 and the last column.

5.4 Loops, for and while

for

A very powerful part of any scripting language is the use of loops. Loops allow one to easily program repetitive tasks. One of the loops commonly used in Bash scripts is the for-loop. A basic for-loop looks like this:

```
for var in range; do
  commands
done
```

Here, range lists the items that var steps through. For each item in range the commands are executed. The easiest way to generate a sequence of numbers is to use the notation {start..stop}. For example

```
for i in {1..22}; do
  echo "This is chromosome ${i}"
done
```

[d] You could have used the same names for both variables, but that's not necessary. I chose two different ones for clarity.

prints 22 messages on the screen. In this case the variable i is increased by one each time the loop is run. In order to use a different step size use {start..stop..step}. As an example, let's print the odd numbers between one and ten on a single line:

```
$ echo {1..10..2}
1 3 5 7 9
```

As you may have already noted, by default files are listed in alphabetical order when using ls. As a result files with numbers in them are not always sorted in the order you expect them:

```
$ ls -lh file*
-rw-r----- 1 lennart lennart  64 2011-01-27 18:08 file1
-rw-r----- 1 lennart lennart  12 2011-08-18 09:30 file10
-rw-r----- 1 lennart lennart  66 2011-01-27 18:08 file2
-rw-r----- 1 lennart lennart   7 2011-08-18 09:29 file20
-rw-r----- 1 lennart lennart  64 2011-01-27 18:08 file3
```

This is easily fixed by using one or more leading zeroes in your file names:

```
$ ls -lh file*
-rw-r----- 1 lennart lennart 64 2011-08-18 09:35 file01
-rw-r----- 1 lennart lennart 66 2011-08-18 09:35 file02
-rw-r----- 1 lennart lennart 64 2011-08-18 09:34 file03
-rw-r----- 1 lennart lennart 12 2011-08-18 09:30 file10
-rw-r----- 1 lennart lennart  7 2011-08-18 09:29 file20
```

In order to use numbers formatted with leading zeroes in for-loops simply add them to the curly braces:

```
for number in {01..22}; do
  echo "This is number ${number}"
done
```

While the {start..stop..step} notation is very handy, you unfortunately cannot use variables in it. For example, the following doesn't work:

```
#! /bin/bash

linecount=$(wc -l ${0} | gawk '{print $1}')

for ln in {1..$linecount}; do
   echo "${ln}"
done
```

seq

For situations like this, the command seq can be used. Its syntax is similar to but not exactly the sames as the {start..stop..step} notation:

```
$ seq 1 10
1
2
3
4
5
6
7
8
9
10
$ seq 1 2 10
1
3
5
7
9
```

So in contrast to the curly braces notation the step increment has to be put in the middle.

Actually, there is a simpler way to loop over files in a given directory. The following example shows how to loop over all csv files in the current directory:

```
for fl in *.csv; do
  echo "File: ${fl}"
```

```
done
```

Although using for-loops for this simple kind of problems where the same analysis has to be repeated a given number of times is very handy, it is not always the best way to do it on modern computers. The essence of the for-loops above is that it runs the analyses one after the other. Modern computers (both servers and desktops) have multiple CPUs, allowing them to run multiple programs at the same time. Consequently, with the serial for-loops above the computer would be seriously underused. There are several ways to make thins more efficient. The preferred method on a compute cluster will be discussed in Chapter 6, which discusses the SGE batch queue system.

Most programming languages not only have for-loops, but also so-called while-loops. Where a Bash for-loop usually runs over a set of variables, a while-loop runs as long as a given condition is true:

```
while condition; do
  commands
done
```

while

The most useful form of the while command is its use in reading input data in a script. Consider the following example: you have a file called genelist that lists the chromosome, start and stop position (in basepairs) of a set of genes, one line per gene. The three values on each line are separated by spaces. The problem at hand is that you would like to extract these regions from a VCF file[e] for further analysis. VCF files can be queried, modified, etc. using the VCFtools programs[f]. The command to extract the region starting at base position 123 400 and ending at position 223 400 on chromosome 1 is:

```
$ vcftools --vcf myVCFfile.vcf --chr 1 --from-bp 123400 \
   --to-bp 223400
```

To automate this extraction for each of the lines in our genelist file we could write a for-loop that extracts each line, uses e.g. cut to put the various columns in variables and proceeds to run the vcftools command.

[e] VCF files are often used to store next-generation sequencing data.

[f] http://vcftools.sourceforge.net

while
read

However, using while in combination with the read command this becomes much easier. Let's create the following Bash script and save it as extract_genes_vcf.sh:

```
1 #! /bin/bash
2 while read chr start stop; do
3   vcftools --vcf myVCFfile.vcf --chr ${chr} \
4     --from-bp ${start} --to-bp ${stop}
5 done
```

Line 2 is where the "magic" happens. The read command waits for the user to type text and then it fills the variables following read with whatever the user typed (spaces separate the values). For example

```
$ read var1 var2
$ echo "${var2} ${var1}"
```

waits until the user has typed two words, puts the first word in var1 and the second one in var2 and then uses echo to print them in reverse order.

So in our extract_genes_vcf.sh script we use while to keep reading three values from the command line and then run vcftools to extract these regions. Of course it would be stupid to type each line from the genelist file by hand on the command line. This is where input redirection comes to the rescue. Remember from §3.13 that we can use < to send the contents of a file to the input of a command. That is exactly what we want here. Instead of running

```
$ ./extract_genes_vcf.sh
```

and having to type all the chromosomes and start and stop positions by hand, we run

```
$ ./extract_genes_vcf.sh < genelist
```

and everything goes as smooth as butter.

Exercise E5.4   For-loops**

a) Write a script that uses a for-loop to run the fictitious command `analyse` for a set of files called `chr1.dat` through `chr22.dat`. However, we want to start at 22 and end with 1, because chromosome 22 is the smallest, so we'll quickly have some results to work with.

b) If you haven't done so already, modify the script in such a way that the numbers in the sequence have a leading zero where necessary. Having files like `chr01.dat`, `chr02.dat` etc. make sure that files are ordered correctly when listing them (`chr02.dat` follows `chr01.dat`, whereas `chr2.dat` follows `chr22.dat`). If you want to see the difference, use the `touch` command instead of our fictitious `analyse` command. `touch` simply creates an empty file.

`touch`

Exercise E5.5*   parallel for-loops**

As mentioned in the text, the for-loops created so far are serial in nature, i.e. the tasks within the loop are started one after another and each task waits until the next one has finished.

a) Using the information on background processes, as described in §3.11, can you think of a way to parallelise the tasks of a for-loop? Write a small piece of example code.

b) Assuming you haven't taken the number of CPUs into account in your previous answer, How take into account that you probably have more tasks than CPUs? You don't need to write code here. Just think how you might achieve this.

5.5 if-clauses and tests

In order to make decisions in a Bash script the `if`-clause can be used. If-structures are important when writing programs and scripts because they allow your program to do different things depending on e.g. input or the output of a certain command.

Here is a basic if-clause in which the value of a variable is tested:

```
1 #! /bin/bash
2 # A simple if-test
3 var="no"
4 if [ "${var}" = "yes" ]; then
5   echo "The value of variable var was yes"
6 fi
```

In line 4 the test is written between square brackets. The value of the string "${var}" was compared to the string "yes" to test whether they are equal. Note the spaces after the opening bracket and before the closing bracket, they are mandatory. Also note the semicolon before the then keyword. As in the case of the semicolon in front of the do keyword in a for-loop it is mandatory.

When writing scripts that accept arguments on the command line, it is good practice to check the arguments before starting the real work. The variable ${*} is the list of all arguments and the variable ${#} contains the number of arguments given on the command line. The following script checks whether the number of arguments is correct, if not it exits with an error message. If the number of arguments is correct it lists all the arguments.

```
#! /bin/bash
# This script demonstrates the use of if tests for
# command line arguments of the script.

n_args=3

# Check if enough arguments are given, else exit the script.
if [ ${#} -ne ${n_args} ]; then
   echo "${n_args} arguments need to be specified, you gave ${#}."
   echo "Exiting..."
   exit
fi

# Print all arguments:
for arg in ${*}; do
   echo "Argument: ${arg}"
done
```

This is what happens if you execute the script with different numbers of arguments:

```
$ ./check_args.sh a b c d e
3 arguments need to be specified, you gave 5.
Exiting...
$ ./check_args.sh a b
3 arguments need to be specified, you gave 2.
Exiting...
$ ./check_args.sh a b c
Argument: a
Argument: b
Argument: c
```

You may have noticed that for the comparison in line 8 `-ne` was used as the "not equal to" operator instead of the `!=` from the first example. In Bash comparison of strings (of text) is handled differently than comparison of numbers. Instead of comparing two strings or two numbers it is also possible to test whether a certain file or directory exists, is writable, is executable, etc. Table 5.1 on page 100 lists all comparison operators for strings, numbers and files/directories.

Exercise E5.6*** **if-clauses and tests**

a) Write a script that tests if the script was run with a command line argument. If that was the case, print the argument. Otherwise print a goodbye message.

b) Write a script that checks if a directory with the name of today's date exists. If not, it should create it.

5.6 Arrays in Bash

Like most, if not all, scripting and programming languages, Bash has the concept of arrays. Arrays are ordered lists, and for those versed in R,

Strings:	
equal	`=`
not equal	`!=`
string `s1` is not empty	`-n s1`
string `s1` is empty	`-z s1`
Numbers:	
equal	`-eq`
not equal	`-ne`
less than ($<$)	`-lt`
greater than ($>$)	`-gt`
less than or equal to ($\leq$)	`-le`
greater than or equal to ($\geq$)	`-ge`
Files and directories:	
Check for directory existence	`-d directory`
Check for file existence	`-e file`
Check for regular file existence not a directory	`-f file`
Check if file is a readable	`-r file`
Check if file is writable	`-w file`
Check if file is executable	`-x file`

Table 5.1: Operators for comparison in Bash. To check for the opposite of the file and directory tests, add a `!` before the test, e.g. to check if a file does not exist use: `if [ ! -e some_file ]`.

arrays are similar to R's vectors. Arrays may contain either numeric or text (string) data.

Initialisation of an array, i.e. the creation of an array is done in a way that is very similar to regular assignment of variables:

```
arr[index]="value"
```

For example, the following example creates an array with student names:

```
students[0]="Alicia"
students[1]="Tim"
students[2]="Pauline"
students[3]="Xue"
```

Note that Bash arrays are zero-based: the index starts at zero[g]

Alternatively, arrays can be initialised in one command like this:

```
students=(Alicia Tim Pauline Xue)
```

Using elements from an array is similar to using regular variables, you simply surround the element with `${...}`, like this `${arr[index]}`. The following example prints the name of the third student:

```
$ echo "Name: ${students[2]}"
Pauline
```

To use the full array use either `@` or `*` as index: `${arr[@]}` or `${arr[*]}`, for example:

```
$ echo "All students: ${students[*]}"
Alicia Tim Pauline Xue
```

The # symbol is used to get the length or size of an array: `${#arr[@]}`, for example:

[g] This is something to keep in mind whenever you are programming. Some languages are zero-based (e.g. C, C++, numpy), others (e.g. R, FORTRAN) are one-based. This difference can lead to so-called off-by-one errors.

```
$ echo "the nr of students is: ${#students[@]}"
the nr of students is: 4
```

The size of a single array element can be found like this `${#arr[index]}`. So if we want to know how many characters are in the third element of the `students` array, we can do something along the following lines:

```
$ i=2
$ echo "the length of ${students[${i}]} is ${#students[${i}]}"
the length of Pauline is 7
```

The simplest way to extend an array is very similar to the way we initialised an array in one command (see above): `arr=(${arr[@]} new1 new2)`, so the following adds two new names to the student array:

```
$ students=(${students[@]} Ivet Lauren)
$ echo "The students now are: ${students[@]}"
The students now are: Alicia Tim Pauline Xue Ivet Lauren
```

The most useful place to use arrays is in for-loops (see also §5.4 on page 99):

```
$ for st in ${students[*]}; do
>   echo "Student ${st}"
> done
Student Alicia
Student Tim
Student Pauline
Student Xue
Student Ivet
Student Lauren
```

5.7 Dealing with errors in your script

Writing bug-free scripts is everybody's goal, but in real life this usually doesn't happen automatically. The following can help to detect bugs.

One of the ways to improve Bash scripts in order to be warned about errors early on is by adding the following line as the top of your script (below the `#!/bin/bash` line and the introductory comments) as the first command to be executed:

`set -e`

```
set -e
```

By adding this command the script will stop with an error as soon as a command in the script finishes with an error. Without this command the script will continue until the end, even when one (or more) of the commands in the scripts gives an error.

There is one case where `set -e` is not enough to stop a script if a command fails with an error, and that is when the erroring command is part of a series of commands joined by pipes (the | symbol; see § 3.13.2). In such a case the script will only exit with an error if the last command of the pipeline fails. To abort the script also if one of the steps in a series of commands fails, add the following to your Bash script:

`set -o pipefail`

```
set -o pipefail
```

Another helpful option to set early in your script is:

`set -u`

```
set -u
```

This options aborts the script when an uninitialised variable is used. Without setting this option a script will continue and simply fill in an empty value when an uninitialised variable is found. For example, take the following script:

```
#!/bin/bash
# A test script for set -u
set -u

echo "The value of the variable var is: ${var}."
```

In the script the variable `${var}` is not initialised and without the `set -u` option the script would simply print

```
The value of the variable var is: .
```

With the `set -u` option set, the following error will be shown:

```
bash: var: unbound variable
```

Of course, this example isn't a huge bug, but imagine a line where the command would be a copy command:

```
cp files.* ~/${destdir}
```

If the `${destdir}` variable is not set before, it will copy all files to your home directory instead of in a subdirectory. This will leave a lot of mess. Or, even worse, consider a similar line where you would remove files from `~/${destdir}`. If the variable isn't set, you would remove all files in your home directory!

If you try to debug your script, running it in the following way will be helpful:

```
$ bash -x my_script.sh
```

This will print every command that is executed, including the contents of variables, etc. Testing your script like this and setting the two options above will be a big help in achieving the goal of writing bug-free scripts.

6

Chapter 6

Working with the SGE queue system

Many, if not all, scientific compute clusters use some system to distribute compute jobs across their nodes. Even if your research group only has a single server it is very worthwhile to have such a system installed. This kind of functionality is not used by default on Linux servers, and consequently this chapter is not generally applicable to other Linux servers. However, most servers or clusters in use in bioinformatics (and other fields with computationally intensive tasks) have such a system.

One of the most commonly used job queuing systems is the Sun Grid Engine (SGE) system, which will be discussed in this chapter. The commands for the PBS system, also very commonly used, are very similar.

On the epib-genstat cluster at the ErasmusMC, regular tasks (i.e. programs not using the SGE queues) are killed after 10 minutes. So it is usually of little use to start an R session or run ProbABEL [6] directly from the command line.

After submitting a job to SGE it will be processed in a so-called queue. Each queue has a certain number of slots. If there are more jobs than slots in a queue the excess jobs will have to wait for a slot to become available (cf. § 6.2 to find out the number of slots per queue or to see whether your job is already running).

Each queue has specific properties and SGE uses these properties to decide which queue to send your job to. In this cluster the queue named `all.q` will be used most of the time.

6.1 Submitting jobs to the SGE queues

Suppose you want to run a certain R script `myscript.R`. Normally you would either start R and then

```
> source("myscript.R")
```

or from the Linux command line you would run:

```
$ R --vanilla -q -f myscript.R
```

6.1.1 Quick and dirty

The quickest way to submit such a task to an SGE queue is the following:

qsub

```
$ qsub -cwd -b y R --vanilla -q -f myscript.R
```

Here qsub is the command to submit a job to the queue system. It is followed by zero or more options (two in this case, -cwd and -b y) and finally the actual command you want to run. The first option, -cwd, stands for “use the current working directory” (this is the same as the present working directory (pwd) mentioned in § 3.6). It tells SGE to look into the pwd for the files you specify (myscript.R in this case) and to write its output files there as well. The option -b y tells SGE that the command you want to execute is not a script but a binary program (R in this case).

Each job that is sent to the queues receives a unique ID, the job ID. This is useful for distinguishing between several jobs you might have waiting in the queue, but is also necessary when you want to delete a job from the queue (cf. § 6.3).

By default SGE will create two files for each job in the queue, one that contains the output that would normally appear on the screen and one that contains the errors that would normally be sent to the screen. These files will have a name that starts with the name of the command you sent to the queue followed by a period, the letter o or e for output and error, respectively, and finally the job ID. For the R command that we submitted earlier the files would be called

```
R.o2823
R.e2823
```

(where of course the number at the end is the job ID, which will be different in your case).

6.1.2 Using a submission script

The preferred way to send a job to the queue system is by using a submission script. Using a script has several advantages:

- you don't have to remember all the command line options for the qsub command, you simply copy your submission script from the previous time you used it (or this website) and only change the program that you want to run.
- All the standard Linux shell scripting tricks are at your disposal.

  A simple example of a submission script for the example of the R script myscript.R used earlier would be:

```
#! /bin/bash
# This is a sample submission script. Lines starting with # are
# comments. The first line (with #!) should be in every script.

# Let's set some variables for SGE. Lines starting with #$ are
# interpreted by SGE as if they were options to the qsub command
# (don't remove the # from the lines starting with #$).
#$ -S /bin/bash
#$ -cwd

# This is the command we would like to run in the queue
R --vanilla -q -f myscript.R
```

  Save this submission script to the same directory as where myscript.R is located and call it for example job.sh. Make sure the script is executable by running

```
$ chmod a+rx job.sh
```

  Now it can be submitted to the queues like this:

```
$ qsub job.sh
```

6.1.3 Refinements to the submission script

The script presented in the previous section is simple but sufficient for basic tasks. Here we present some additions to the script that can make life with SGE easier. A script file with all the suggested options can be downloaded from http://epib-genstat.erasmusmc.nl/qscript.sh. The only things that need to be changed are the e-mail address and the last line where you fill in the command(s) you want to run.

- Start/stop e-mails
  Since most jobs will take more than 10 minutes to complete (otherwise you could have run them without using the queue system, right?!) it would be nice to get an e-mail when the job is finished so that you don't have to run qstat all the time (cf. § 6.2). To get an e-mail when a job begins and when it ends simply add the following two lines after the #$ -cwd line in the aforementioned simple script:

  ```
  #$ -M your_address@your_domain.com
  #$ -m be
  ```

  If you only want an e-mail after the job has finished change the #$ -m be option to #$ -m e.

- Output to a single file
  As discussed at the end of section 6.1.1 the output and error messages of a job are recorded in separate files. When running a large amount of jobs this can lead to a proliferation of these files. By adding

  ```
  #$ -j y
  ```

  to your submission script the output and error files will be joined into one file of the form script.o2345.

6.2 Monitoring progress

qstat

The qstat command allows you to see whether your job is accepted by one of the queues, which jobs you have submitted so far, how many jobs are waiting in the queues, etc. Simply running

```
$ qstat
```

qstat -f

will show your own running and waiting jobs. Running the command

```
$ qstat -f
```

will give an overview of all queues, even the ones in which you don't have any jobs running. To show all jobs of all users in all queues use

qstat -f

```
$ qstat -f -u \*
```

This gives you an idea how busy the cluster is. A sample output on a quiet day looks like this:

```
queuename                      qtype resv/used/tot. load_avg arch      states
-------------------------------------------------------------------------
all.q@node01.polyomica.com     BP    0/2/7          2.03     lx24-amd64
   2843 0.56000 R          user1        r     08/06/2010 10:16:42     1
   2844 0.56000 R          user1        r     08/06/2010 10:16:42     1
-------------------------------------------------------------------------
high_prio_q@node01.polyomica.c  BP    0/0/4          2.03     lx24-amd64
-------------------------------------------------------------------------
int.q@node01.polyomica.com     IP    0/0/2          2.03     lx24-amd64
```

It shows the user user1 has two jobs in the queue called all.q with job IDs 2843 and 2844. Both jobs are running some R script. The queue called high_prio_q is empty as is the queue called int.q, which is used for interactive jobs only (because the q-type has the letter I, queues that accept batch jobs (i.e. regular commandline programs or shell scripts) have type B). The resv/used/tot. column show the number of reserved and available slots in each queue as well as the total number of available slots. So in this example all.q has no reserved slots and two slots are in use out of a total of seven. If there are more jobs than slots in a queue the excess number of jobs will have to wait until slots become available again.

6.3 Deleting jobs from a queue

At some point you will find that you want to delete a job from the queue. This may happen because you submitted five R jobs, for example, and the first one finished early because you made a typing mistake in the R code. Since the other jobs use the same R code they will finish with an error as well so you decide to remove them from the queue. For this you use the qdel command followed by the job ID. Use qstat (cf. § 6.2) to find the job ID of your jobs. Running

qdel

```
$ qdel 2844
```

would kill the second job in the list shown in § 6.2. Of course a user can only delete her/his own jobs.

6.4 Getting info on a finished job

qacct

qacct -j

To get information on a job that has finished, use the `qacct` command in combination with the job ID:

```
$ qacct -j 8765
==============================================================
qname     all.q
hostname  node01.polyomica.com
group     genepi
owner     some_user
project   NONE
department genepi
jobname   probabel.pl
jobnumber 8765
taskid    undefined
account   sge
priority  8
qsub_time Mon Mar 7 22:56:01 2011
start_time Mon Mar 7 22:56:15 2011
end_time  Tue Mar 8 03:59:44 2011
granted_pe NONE
slots     1
failed    0
exit_status 0
ru_wallclock 18209
ru_utime  17362.282
ru_stime  816.388
ru_maxrss 0
ru_ixrss  0
```

```
ru_ismrss 0
ru_idrss  0
ru_isrss  0
ru_minflt 328576642
ru_majflt 2
ru_nswap  0
ru_inblock 0
ru_oublock 0
ru_msgsnd 0
ru_msgrcv 0
ru_nsignals 0
ru_nvcsw  12597
ru_nivcsw 443538
cpu       18178.669
mem       21867.259
io        170.594
iow       0.000
maxvmem   2.037G
arid      undefined
```

The most interesting elements of the output are `start_time`, `end_time`, for the time at which the job started and when it finished, respectively. Note that the submit time has a separate entry. The value of `cpu` shows you the number of seconds of CPU time the job used. The value of `ru_wallclock` shows the total time (in seconds) that the job took (i.e. CPU time, but also time used for reading/writing files etc.). The values `qname` and `hostname` tell you in which queue and o which server the job was run.

In order to find out how much time your jobs have spent in the queue over the last 15 days run (with your own username, of course):

```
$ qacct -o lennart -d 15
OWNER    WALLCLOCK       UTIME      STIME         CPU      MEMORY          IO ↩
           IOW
=========================================================================
lennart      96279   10798.155   1903.326   12701.481    3529.165    4165.959 ↩
         0.000
```

The `WALLCLOCK` time is the total time in seconds that your jobs have spent running in the queue (in the last 15 days). The CPU column shows the time (in seconds) the job was actively using a CPU, i.e. not waiting for other things like reading or writing a file (which is listed in the IO column).

6.5 Interactive jobs

Although it is not the preferred way to run programs, sometimes it may not be possible or efficient to write a complete script to submit to the queue. For example, you'd like to start an R session and enter the commands on the R command line because you are not sure whether a certain construction works on the cluster.

In such a case the interactive queue (int.q) can be used. Starting a session in the interactive queue is done like this: qrsh

```
$ qrsh -pty y command
```

where command is the program you'd like to run (R or solar are likely examples). You will then be asked for your password and an interactive session is started on either one of the servers in the cluster (if there are slots available in the interactive queue of course).

Note that jobs running in an interactive queue will be killed after 48 hours of CPU usage and they will run with a very low priority.

If, for some reason you can't access your interactive queue session anymore, you can kill the old one using the qdel command.

6.6 Exercises

Exercise E6.1** Working with Sun Grid Engine

In this exercise you will create a small script and submit it to the SGE batch system. You will monitor its progress and then use a Bash script to automate submission to SGE.

a) Create a new directory in the directory ~/LinuxCourse/ you created in Exercise E3.3.

sleep

b) The sleep command accepts one argument: the time to wait in seconds. The whoami command prints your user name. Write a Bash script that prints your user name and the current time (including seconds), then goes to sleep for 10 seconds and repeats this cycle 5 times. End the script with a final print of the current date and time, followed by a line of dashes. Run the script to test whether it works.

c) Now submit this script to the queue. Don't forget to add the -cwd option, otherwise the files generated by SGE will end up in your home directory. Check to see that the script is either running or waiting to be run. If your script is waiting, what command would you use to find out how busy the batch system is?

d) After the script has run, inspect the two resulting output files.

e) Write a Bash for-loop[a)] to submit five instances of this script to the queue. Monitor them and see how jobs of different users are scheduled.

Exercise E6.2** SGE, R and command line arguments

Exercise E5.3 on page 90 explained how to use command line arguments in Bash scripts. Wouldn't it be fun if the could be done in R scripts as well? This exercise will show you how.

To run an R script from the command line (so not in interactive mode), use the -f option to specify which script to run. Usually the options -q (for 'quiet', to suppress the startup message), --slave (to be really quiet) and --vanilla (see R --help for more info on this option) are specified as well. For example:

```
$ R --vanilla --slave -q -f myscript.R
```

[a)] You don't necessarily need to do that in a script. Simply writing a for-loop on the command line is also OK.

To pass arguments to the R script use the `--args` option as the last option, followed by any arguments you want to send to the script. The number `1234` and the string `"Hello there"` can be passed like this:

```
$ R --vanilla --slave -q -f myscript.R --args 1234 "Hello there"
```

In order to use these arguments in an R script, use the `commandArgs()` function like this:

```
args <- commandArgs(TRUE)
```

This stores all the arguments following the `--args` option in the array `args`. The following R script gives you an idea how to use all this.

```
cat("Welcome to this simple R script.\n")
cat("These were the command line arguments:\n")
args <- commandArgs(TRUE)
args

cat("Now we will print the arguments one by one:\n")
for (i in 1:length(args)) {
  cat(i, ": ", args[i], "\n", sep="")


# All arguments are treated as strings (text), so in
# order to use them in calculations they have for be
# converted to numbers:
number = as.integer(args[1]) + 2
cat("number = ", number, "\n")

cat("The end\n")
```

a) Write an R script that accepts command line arguments. The first argument is a start value, the second one is a stop value (both are integer numbers). The R script should use these values as start and stop values in a for-loop. Since this example is purely didactic, you can do something simple inside the loop, like printing the square of the loop value. Verify that it works as expected.

The quick and dirty way to submit this script to SGE would be

```
$ qsub -cwd -b y R --slave --vanilla -q -f R_loop_script.R --args 1 5
```

And if you want to join the output of SGE (the .e and .o files) into one as well as receive an e-mail when the job starts and when it ends, you will have an even longer command:

```
$ qsub -cwd -b y -j y -m be \
    -M your_email@your_domain.com \
    R --slave --vanilla -q -f R_loop_script.R --args 1 5
```

This is too much to remember or type without making errors. The solution is to write job scripts for tasks like this. In a job script you specify all the SGE options you need as well as the command that is to be run in the batch queue. Then, the next time you run a similar task you only need to change the line that runs R (or whatever else you would like to run in the queue).

To specify one of the SGE options in a job script (which is a normal Bash script) you have to start the line with #$. The following script is an example job script for the R script of this exercise. The -S option in line 5 makes sure that SGE understands this is a Bash script.

```
 1  #! /bin/bash
 2  # This is an example of a job submission script
 3
 4  # The following are options for qsub. Don't
 5  # remove the # in front of them.
 6  #$ -S /bin/bash
 7  #$ -j y
 8  #$ -cwd
 9  #$ -M your_email@your_domain.com
10  #$ -m be
11
12  # Here comes the rest of your script that
13  # actually does something. In this case it runs R.
14  R --slave --vanilla -q -f R_loop_script.R --args 1 5
```

With this script submitting a job to SGE is simple:

```
$ qsub jobscript.sh
```

b) Rewrite this example job script in such a way that it accepts two command line arguments and uses those in the R command, instead of the 1 and 5. Add some checks to see if the user really did specify two command line arguments. If that is not the case, the script should exit (use the `exit` command for that). Test the script by submitting it to the queue.

Chapter 7

Good scripting practices, structured programming and data management

By now you should know your way around basic Bash scripts. Some of you will also have done some scripting in R as well. Here I would like to discuss some general principles that relate to good programming and scripting[a] practices and how those relate to reproducible research.

In most sciences it has become impossible to work without writing some form of computer code. Scripts help you analyse your data, run simulations, etc. and form an integral part of your research. A person in a wet lab has his/her lab journal in which each step of the protocol is meticulously written down, and so do you have your directory structure, your data sets and your scripts. As a scientist it is your responsibility to make sure that you can always reproduce your results. This is one of the fundamentals of the scientific method.

Some of the following tips may seem obvious, most will take a bit of extra time to implement. But rest assured, they are all worth it and have proven themselves in practice. Most of them boil down to using descriptive names and comments in your code and directory structure. Remember that some projects take years to complete, and some get handed over to colleagues.

Resist the temptation to skip the tips in this chapter because “this script will only be used once”. Many scripts survive longer than that! And good scripts will continue to be useful to you and maybe other people. They will be used again and again, sometimes with some slight modifications. What is the point of writing similar scripts from scratch all the time? If you run into a situation similar to one you have encountered before it

[a] A quick note about the words scripting and programming. Although they may often be used interchangeably, at least in our field, there is a difference. Examples of scripting languages are Bash (cf. § 5), R, sed (§ 4.3), gawk (§ 4.5) and Perl. Java, C, C++ and FORTRAN are examples of programming languages. The difference between the two groups is that scripting languages are interpreted on the fly. This means that the are converted from human-readable language (your code) to machine language the moment you run them. Code written in a programming language is converted to machine language once (this step is called compilation) and can then be run many times. As a result, programs written in a scripting language are usually slower to run, but easier to write and debug. Apart from that, there is not much difference between the two in terms of writing code.

is better to extend and generalise an existing script than to try and invent the wheel again. Furthermore, well written scripts with a good code layout help you find mistakes (so-called debugging) much easier, saving you a lot of time in the process.

7.1 Code layout

As Bradnam and Korf write in their chapter on code beautification: “Appearance matters” [8]. This is definitely true! Scripts in which the code is laid out well are easier to understand and consequently easier to debug. Consider the following examples of a piece of code shown earlier:

```
#! /bin/bash

if [ -n "${1}" ]; then
   echo "The command line argument was ${1}"
else
   echo "Goodbye!"
fi
```

and

```
#! /bin/bash
if [ -n "${1}" ]; then echo "The command line argument was ${1}"; else echo ↩
    "Goodbye!"; fi
```

Both are valid Bash code that produce the same output, but in terms of understanding and maintaining the code, the first version definitely has the upper hand.

7.1.1 Indentation

Indenting the lines of code in the `if` and `else` part of the previous example clearly shows what the code is supposed to do. A good editor, like Emacs or (g)vim helps you with the indentation. In Emacs, for example, pressing the TAB key in a Bash or R script will automatically indent that line correctly. Whether you choose to indent with a TAB, four spaces (most common) or only two is not very important, as long as you are consistent. A good editor also helps you by colouring the various parts of the code, as shown here[b]:

```
#! /bin/bash
# This is a comment.

if [ -n "${1}" ]; then
   echo "The command line argument was ${1}"
else
   echo "Goodbye!"
fi
```

Note that in this syntax highlighting scheme comments are shown in green, Bash keywords in blue and strings in purple. If, for example you forget a quote, syntax highlighting will immediately show you something is wrong:

```
#! /bin/bash
# This is a comment.

if [ -n "${1}" ]; then
   echo "The command line argument was ${1}
else
   echo "Goodbye!"
fi
```

7.1.2 Line length

In the early days of computing the maximum length of a line was 80 characters[c]. Many programmers still stick to this limit, and with good reason. Consider the following VCFtools command:

```
vcftools --vcf myVCFfile.vcf --chr ${chr} --from-bp ${start} --to-bp ↩
    ${stop} --plink-tped --recode --out geneselection
```

[b] For those of you reading the PDF version of this document on screen or a colour print.

[c] That was the width of the punch cards used to program a computer in those days.

I hope you are convinced the following is easier to read (remember that Bash needs a backslash (\) at the end of a line that hasn't finished yet):

```
vcftools --vcf myVCFfile.vcf --chr ${chr} \
   --from-bp ${start} --to-bp ${stop} \
   --plink-tped --recode --out geneselection
```

and some of you would like the following even better:

```
vcftools --vcf myVCFfile.vcf \
         --chr ${chr} \
         --from-bp ${start} \
         --to-bp ${stop} \
         --plink-tped \
         --recode \
         --out geneselection
```

The same holds for R code, of course. Compare

```
data <- read.table("/path/to/file", header=TRUE, ↩
   sep=";", na.strings="0", stringsAsFactors=TRUE)
```

to

```
data <- read.table("/path/to/file",
                   header=TRUE,
                   sep=";",
                   na.strings="0",
                   stringsAsFactors=TRUE)
```

7.1.3 Spaces

Clean code makes good use of white space. Although Bash doesn't allow for a space between the variable name, the = sign and the value, other languages do and it is good practice to surround your equal signs with a space. The same holds for commas, semi-colons and pipes (|). Compare the following lines of Bash code:

```
linecount=$(wc -l ${0} | gawk '{print $1}')
linecount=$(wc -l ${0}|gawk '{print $1}')
```

or the following GAWK one-liners:

```
$ gawk 'BEGIN{print "id sex bt1"}{if($5==2)sex=0;if($5==1)sex=1;print ↩
    $2,sex,$6}' chr.ped > chr.phe
$ gawk 'BEGIN {print "id sex bt1"} {if($5==2) sex=0; if($5==1) sex=1; print $2, ↩
    sex, $6}' chr.ped > chr.phe
```

or this piece of R code from which I intentionally removed the colours used for syntax highlighting:

```
f2dna<-read.csv(paste(dir,"dna_f.csv",sep=""))
## Read data from Excel sheet
library(gdata)
dbFile<-paste(dir,"db.xls",sep="")
db<-read.xls(dbFile,sheet=1)
## Create a new data frame that combines the old data with
## selected columns from the DB
cols<-cbind("Library","Sample.ID","Cov.X")
selectedData<-olddata[,cols]
cols<-cbind("nummer","trait","finaldiag")
selectedDataDB<-db[which(db$nummer %in% ↩
    selectedData$Sample.ID),cols]
```

versus

```
f2dna <- read.csv(paste(dir, "dna_f.csv", sep=""))

## Read data from Excel sheet
library(gdata)
dbFile <- paste(dir, "db.xls", sep="")
db     <- read.xls(dbFile, sheet=1)

## Create a new data frame that combines the old data with
## selected columns from the DB
cols           <- cbind("Library", "Sample.ID", "Cov.X")
selectedData   <- olddata[, cols]
```

```
cols           <- cbind("nummer", "trait", "finaldiag")
selectedDataDB <- db[which(db$nummer %in%
                           selectedData$Sample.ID),
                     cols]
```

and decide for yourself.

Space in the form of empty lines can also be very helpful in understanding a script. Use empty lines to separate parts of your script that do different things, as shown in the R example above.

7.2 Comments

All scripts should have a header that explains the intended use. Right now it is obvious what the script does (or is supposed to do), but in six months time you will look at it again and guessing what a script does simply from its file name or the directory it was in takes much more time than writing a brief description.

It is good practice to also add your name and a date to the header. Good scripts provide added value, also to other people. They tend to get distributed. It's good to be able to tell who wrote the original script and, if you run into two versions of a script, which one is newest[d].

If your script accepts command line arguments describe them in the header. This helps you if you need to run the script at a later time.

Write comments that describe the more intricate parts of your script. They will help you (or you successor) to understand what (is supposed to) happen(s). Especially if it took you quite some time to come up with an elegant solution to a problem having a comment that describes it works miracles when troubleshooting later on.

You can also use comments to separate parts of your code. Especially with syntax highlighting they will stick out from the rest of the code.

[d] If you are interested in a more professional and efficient way of storing different versions than simply adding a version number to the header or the file name, see the item on revision control in Chapter 8.

```
#-------------------------------
# Initialise all variables here
#-------------------------------

##
# Read data from files
##

##########################
# This is the main part
##########################
```

7.3 Variable names

Some people are tempted to use variable names like `a`, `b`, `aa`, `thingy`, `df`, etc. These will make your life miserable! Looking back at a script in a year's time names like `ERFgenotypes`, `IDs_no_meds` or `HM2_snpnames` are much more descriptive.

Try to find the balance between short non-descriptive variable names and ones that are too long and only lead to typo's. Try using underscores (_), CamelCase, or periods to make variable names descriptive and easy to read.

7.4 File and directory names

A well-organised directory tree is like a well-kept house. You immediately know where to find things. Try to think of a logical structure for your directories, e.g. along projects, data sets, etc. Give the directories informative names. Don't be worried about having to type long names. You haven't forgotten TAB completion, have you? Also allow yourself some time to clean your directories, just like most people do with their desks before going on holiday.

It is also good practice to add a file called README to each directory in which you briefly describe what you do, did, or plan to do there.

Give your output files sensible names. Having a directory with the following files is not very helpful (even one week from now).

```
output.1
output.2
output.data1.2
output.try2.txt
```

If you argue that the file with the latest date/time is the one you should use, how can you be sure? What if you actually didn't completely finish the analysis? What if the output with the latest date was actually a run that was discarded because you were trying an option that turned out not to be correct?

7.5 Summary

- Begin your scripts with a header that documents its behaviour.
  - Start your script with a comment that summarises the use of the script.
  - Add your name and the date you last edited the script to the header.
  - If your script needs (or accepts) command line arguments, give a brief description of each of them in the header.
- Comments are good!
- Give variables a descriptive name.
- Split long lines into shorter ones.
- Use white space wisely.
- Create a new directory for each project.
- Give your output files a descriptive name.

- Be wary of massive scripts. As soon as scripts grow beyond, say, the size of your screen, consider splitting parts off into separate scripts. Or learn about the use of functions[e]. Again, this will help you structure your code.
- Don't ignore errors and warnings. Most of them are there for a reason. You might consider them a nuisance, but make sure you understand what they mean and where they come from or your research results may be invalid.
- Don't start tomorrow. Start following these guidelines today. Tomorrow there will be other high priority things.

[e] In other languages like Perl functions are known as subroutines.

8

Chapter 8

Where to go from here?

The course for which this document was written was a two-day course. If you have come this far within these two days then either you had quite some Linux experience already or you picked it up really quickly, on which I have to congratulate you!

8.1 More advanced topics

If you are interested to learn more, you can consider looking into the following subjects (GIYF, but I'll be happy to give you some hints):

byobu
screen

- Using `byobu` or `screen` to access the same connection from multiple locations; With `byobu` or `screen` it is possible to continue working with the same shell session from multiple locations.
- Shell expansions; In § 3.6.3 the use of wildcards was explained, where characters like `*` were used to select multiple files. Using so-called shell expansion allows you to work even more efficiently on the CLI.

find

xargs

- Finding files and directories with the `find` utility. The syntax of the `find` utility is not the easiest to understand, but using the basic ones together with the `xargs` command makes for a powerful combination if you want to apply the same action to many files.
- Regular expressions; The search patterns of `sed`, `gawk`, `grep` and other utilities are not just literal searches. In fact they are so-called regular expressions. With regular expressions it is possible to do a search like "Find every occurrence of *word*, but only if it is located at the end of a line", or "Find every occurrence of *p*, but only if it is followed by several numbers, followed by an `e`, then a `+` or `-` and a maximum three digits".
- More advanced Bash scripting; The use of case structures, how to let a script accept options in short and long form, making functions, etc.

- More advanced GAWK scripting; Making functions, associative arrays, using other values than the newline character as a record separator, etc.
- Revision control; If you write scripts on a regular basis and re-use them regularly, you will find that at some point you have quite a library and maybe you want to reduce their number by reorganising them or making them more generalisable. A great idea, but wouldn't you want to be able go back a couple of versions every once in a while to find out why something used to work, but doesn't anymore? Of course you could save every version with a different file name, but that would make a big stack of files. With revision control files (or complete directories) can be saved, easily shared with others and you will always be able to find out what changed between revisions. Several programs provide version control. Subversion (`svn`) is very commonly used, but slowly getting old and superseded by Git (`git`) [9]. Bazaar (`bzr`) is easier to start with. All of these provide good integration with MS Windows through a GUI as well. GitHub (`https://github.com`) and BitBucket (`https://bitbucket.org`) are popular web services where you can collaborate with other people on the same project using Git. More and more open source tools developed by scientists can be found there.
- For those of you that like to learn another scripting language I can suggest both Python and Perl. Python is the most "general purpose" of the two, meaning that you can not only use it for automating tasks, but also to do scientific analyses, or even write GUI programs. Perl is traditionally more focused on text processing and used a lot in bioinformatics.

8.2 Further reading

One of the most appropriate books on this subject is *UNIX and Perl to the Rescue* by Bradnam and Korf [8], two bioinformaticians at UC Davis. This book also teaches the basics of the Perl scripting language which is often used in bioinformatics. If you would like to learn more on Bash scripting, take a look at Refs. [10, 11]. To learn more about general programming concepts, try Ref. [12] (it does require some knowledge of the C programming language to really understand the text). Ref. [13] is both a good reference and a good course book on Linux system administration (in case you want to set up your own Linux server), since it trains you for the LPIC-1 exam of the Linux Professional Institute. Those of you who want to learn how to use their Emacs in a more efficient manner can read Ref. [14], which, although a bit outdated (it discusses Emacs 22, whereas Emacs 25 is the latest version), is still a good introduction to the core concepts. More recently, Mickey Petersen wrote a good e-book on Emacs [15].

A

Appendix A

Answers to the exercises

Answer to Exercise E3.1. Long and short options

The `--all` (long form) option lists all files and directories (both normal ones and hidden ones). The second command will also show a list of files and directories. The reason for that is, that `-all` is interpreted as the contraction of the short options `-a -l -l`.

Therefore, the second form, with only one dash is probably a typing mistake where the user either forgot the second dash or the last `l` should have been an `h` (which means show file sizes in human readable format).

Answer to Exercise E3.3. Some file and directory basics

In the following answers I did not include all the output of each command in order to save some space.

Running `cd` without arguments brings you to your home directory. Of course `cd ~` and `cd /home/your_username` are also correct.

```
$ cd
$ mkdir LinuxCourse
$ cd LinuxCourse
$ mkdir tmp2
$ mkdir tmp3
```

Note that the last two lines can be written as one: `mkdir tmp2 tmp3`.

```
$ cd tmp2
$ cd ../tmp3 # or: cd ~/LinuxCourse/tmp3
$ cd
```

The shortcut to go to the previous directory you visited is `cd -`. The `pwd` command will print your present working directory on the screen.

```
$ cd /
$ ls
$ cd /tmp
$ ls
$ cd ../home/lennart/LinuxCourse
```

Of course, you have to replace my user name with yours in the last line.

In the following I did include all the output of the commands.

```
$ ls
tmp2 tmp3
$ rmdir tmp2 tmp3
$ ls
$
```

Answer to Exercise E3.4. Copying files

The last argument of the `cp` command is always the destination, which means in this case that you are trying to copy `file1` and `file2` to another file `file3`, which does not make sense. So, either you forgot to add a destination directory to the `cp` command, or you made a typo and the last argument should not have been `file3`, but the name of the destination directory.

Answer to Exercise E3.5. Creating a directory tree

Both `mkdir --help` and `man mkdir` will tell you that the `-p` option allows to create missing parent directories:

```
$ mkdir -p dira/dirb/dirc
```

Removing directories can be done in two ways. If the directories are empty, the `rmdir` command will do the job. Since `dira` is not empty (it contains `dirb` and `dirc`), `rmdir dira` will not work. One solution is then to first go to directory `dirb` and start from there:

```
$ cd dira/dirb
$ rmdir dirc
$ cd ..
$ rmdir dirb
$ cd ..
$ rmdir dira
```

This takes way too much typing. In this case, the `rm` command will help, the option `-r` will remove files and directories recursively, reducing the above to a single (but more dangerours) command (run from the parent directory of `dira`)

```
$ rm -r dira
```

Answer to Exercise E3.6. Getting information on files and directories

On my Linux system I found the following files in `/boot`:

```
$ ls -lh /boot/
-rw-r--r-- 1 root root 983K Oct  9 18:49 abi-3.11.0-12-generic
-rw-r--r-- 1 root root 899K Sep 10 22:29 abi-3.8.0-31-generic
-rw-r--r-- 1 root root 160K Oct  9 18:49 config-3.11.0-12-generic
-rw-r--r-- 1 root root 152K Sep 10 22:29 config-3.8.0-31-generic
drwxr-xr-x 5 root root 4.0K Oct 20 14:51 grub
-rw-r--r-- 1 root root  18M Oct 20 14:50 initrd.img-3.11.0-12-generic
-rw-r--r-- 1 root root  17M Oct 20 14:31 initrd.img-3.8.0-31-generic
drwx------ 2 root root  16K Sep 15 10:47 lost+found
-rw-r--r-- 1 root root 173K Jun 17 11:52 memtest86+.bin
-rw-r--r-- 1 root root 175K Jun 17 11:52 memtest86+_multiboot.bin
-rw------- 1 root root 3.2M Oct  9 18:49 System.map-3.11.0-12-generic
-rw------- 1 root root 3.0M Sep 10 22:29 System.map-3.8.0-31-generic
-rw------- 1 root root 5.4M Oct  9 18:49 vmlinuz-3.11.0-12-generic
-rw------- 1 root root 5.2M Sep 10 22:29 vmlinuz-3.8.0-31-generic
```

So there are two files with `vmlinuz` in their name, one is 5.4 MB, the other is 5.2 MB. Note that on your system files with different version numbers may be used.

The users and groups of the three directories are can be found also using the `ls -l` command. The following lines show user, group and directory name:

```
root  root  crash
root  staff local
root  mail  mail
```

Again, this may be slightly different on your machine. These are the steps I took to get that information (check the 3rd and 4th columns):

```
$ cd /var
$ ls -lh
total 60K
lrwxrwxrwx  1 root root     9 Aug  5  2011 adm -> /var/log/
drwxr-xr-x  2 root root  4.0K Aug 18 06:35 agentx
drwxr-xr-x  2 root root  4.0K Oct 17 06:35 backups
drwxr-xr-x 20 root root  4.0K Feb 21  2013 cache
drwxrwxrwt  2 root root  4.0K Oct 15 06:25 crash
drwxr-xr-x 75 root root  4.0K Jul 31 16:17 lib
drwxrwsr-x  2 root staff 4.0K Apr 21  2011 local
lrwxrwxrwx  1 root root     9 Jun 27  2012 lock -> /run/lock
drwxr-xr-x 19 root root  4.0K Oct 18 06:32 log
drwx------  2 root root   16K Aug  3  2011 lost+found
drwxrwsr-x  2 root mail  4.0K Oct 18 01:34 mail
drwxr-xr-x  2 root root  4.0K Aug  3  2011 opt
lrwxrwxrwx  1 root root     4 Jun 27  2012 run -> /run
drwxr-xr-x 10 root root  4.0K Feb 12  2013 spool
drwxrwxrwt  4 root root  4.0K Sep 21 14:43 tmp
```

A more 'focused' approach would have been to ask `ls` to only show the information for the directories we are interested in (note the `-d` option):

```
$ ls -lhd mail crash local
drwxrwxrwt 2 root root   4.0K Oct 15 06:25 crash
drwxrwsr-x 2 root staff 4.0K Apr 21  2011 local
drwxrwsr-x 2 root mail   4.0K Oct 18 01:34 mail
```

Answer to Exercise E3.7. The dangers of wildcards

The first `rm` command removes all files and directories ending in ~ (usually files ending with a tilde (~) are backup files), the second rm removes all files and directories in the present directory and all (!) files and directories in your home directory. You most likely don't want that.

Answer to Exercise E3.8. Working with less

Searching for “search” or “backward” will show that the `?` key followed by a search pattern will search backwards. Note that on most keyboards this is conveniently located on the same key as `/`.

Answer to Exercise E3.9. ”Seeing a file grow”

Either use the man page or the `--help` option of `tail` to find that the option is `-f` (or `--follow` in the long form).

Answer to Exercise E3.10. Untar-ing an archive

Copying and extracting the file goes like this (assuming you are in the `LinuxCourse` directory):

```
$ cp /tmp/exercises_linux_course.tar.gz .
$ tar -xzf exercises_linux_course.tar.gz
$ ls -p
exercises_linux_course/
```

In the last step the `-p` option was added to explicitely show that `exercise_data` is a directory. Likewise, `ls -l` will show the same because of the `d` in the first permission column:

```
$ ls -l
drwxr-x--- 3 lennart genepi 4096 2010-11-07 23:09 exercises_linux_course
```

Answer to Exercise E3.14. Disk space usage

Follow the example:

```
$ du -sh ~/*
```

Answer to Exercise E3.15. Downloading files to the server

When checking the size of the file, don’t forget to add the `-l` option (or both `-l` and `-h`: `ls -lh` to your `ls` command, otherwise the size information won’t be printed.

Adding the `-c` option to `wget` continues a download, as you can see from reading the output of either of the following commands:

```
$ wget --help
$ man wget
```

Answer to Exercise E3.16. Combining files

The cat command simply prints the contents of a file on the screen. Therefore,

```
$ cat file1 > output.total
```

will send that output to the output.total file. To add the contents of the other files you need to use >>, because > will automatically overwrite the output file if it exists, whereas >> appends to the end of the output file.

```
$ cat file2 >> output.total
$ cat file3 >> output.total
```

The check whether the three files we merged you could try several things. You could quickly browse through output.total with less or more (although that isn't efficient for large files). Or you can count the number of lines for each file and see if they add up (wc -l).

Answer to Exercise E3.17. Combining input and output redirection

Input redirection can simply be combined with output redirection:

```
$ R --no-save --quiet < rinput.R > Routput
$ cat Routput
> print("Hello, you are now in R")
[1] "Hello, you are now in R"
> getwd()
[1] "/tmp"
> 1+1
[1] 2
> 10:1
 [1] 10 9 8 7 6 5 4 3 2 1
>
```

The cat command is only used to check the contents of the Routput command.

Answer to Exercise E3.18. Using the output of one command as input for another

The command ps -u your_username lists all your processes, one on each line. To count the number of lines in the output use wc -l. Combining these we get: ps -u your_username | wc -l. Don't forget that the output of ps has a header, so you have to subtract 1 from the number you got to get the number of processes!

Use du to find the disk usage of each of the files and directories in your home directory. The -h option of the sort commands allows you to sort the output of the du -h command. Without the -s (summarize) option du returns the disk usage of each individual file in each of the subdirectories.

```
$ cd
$ du -sh * | sort -h
```

Notice that the output of w has a two-line header. So

```
$ w | wc -l
```

gives you the number of logged-in users plus two. In order to get the number of unique users the uniq command can be used. However, uniq only removes duplicate entries if they are on adjacent lines. Therefore it is wise to use the sort command first. However, since every line is slightly different (for example each connection gets a unique entry in the TTY column) uniq will not find any unique lines. So, we must select only the first colum (the one with the user names) and sort that before applying uniq:

```
$ w | cut -f1 -d" " | sort | uniq | wc -l
```

Note that we still have to subtract 2 from the output of the line count.

Answer to Exercise E4.1. Converting files from Windows format to Linux format

The command line option to write to a new file is -n. You could have found this out by either of the following commands:

```
$ dos2unix --help
$ man dos2unix
```

Note the remark about the order of the option and the two file names! First you specify -n, then the input file and then the output file.

The following commands run the file command on a text file, convert it from Linux to DOS format and test the new file:

```
$ file file_in_linux_format
file_in_linux_format: ASCII text
$ unix2dos -n file_in_linux_format file_in_dos_format
unix2dos: converting file file_in_linux_format to file ↩
    file_in_dos_format in DOS format ...
$ file file_in_dos_format
file_in_dos_format: ASCII text, with CRLF line terminators
```

Answer to Exercise E4.2. Searching for a given text in a file

If the phenotype file is called file.phe and the individual ID we are looking for is 1234, the command

```
$ grep 1234 file.phe
```

will show us the line if it is present. Be careful! This command will also show lines containing 12341, 12345, a1234, etc. To make sure you only get the person with ID 1234 add the -w option.

```
$ cd ~/LinuxCourse/exercises_linux_course
```

The option for recursive searching is -r (use man grep or grep --help to find this). Therefore, the command

```
$ grep -r trait *
```

will look in all files and directories (starting in your present working directory) for the text “trait”.

To look for single words only, use the -w option:

```
$ grep -rw trait *
```

The options could also have been written as separately: `-r -w`, but `-rw` is shorter.

A case-insensitive search is done with `-i`. The `man`-page will tell you this, as will `grep --help`. A nice way to quickly find this is using `grep` on `grep` itself:

```
$ grep --help | grep case
  -i, --ignore-case   ignore case distinctions
```

Here, we looked for the word “case” in the output of `grep --help`.

This is how to add colour to your life with `grep`:

```
$ grep -ri --color=auto trait *
```

Answer to Exercise E4.3. Using `sed` for search-replace operations

If you followed Exercise E3.10 to the letter, the files should be here:

```
$ cd ~/LinuxCourse/exercises_linux_course/Exercise_sed/
$ ls
file.csv
$ more file.csv
# field1,field2,field3,field4
11,12,13,14
21,22,23,24
31,32,33,34
41,42,43,44
51,52,53,54
61,62,63,64
```

This replaces all comma’s with tabs:

```
$ sed 's/,/\t/g' file.csv
# field1 field2 field3 field4
11 12 13 14
21 22 23 24
```

```
31 32 33 34
41 42 43 44
51 52 53 54
61 62 63 64
```

To send the output to a file use output redirection:

```
$ sed 's/,/\t/g' file.csv > file.tsv
$ more file.tsv
# field1 field2 field3 field4
11 12 13 14
21 22 23 24
31 32 33 34
41 42 43 44
51 52 53 54
61 62 63 64
```

Editing the original file directly is easy with `sed`'s `-i` option:

```
$ sed -i 's/\t/;/g' file.tsv
$ more file.tsv
# field1;field2;field3;field4
11;12;13;14
21;22;23;24
31;32;33;34
41;42;43;44
51;52;53;54
61;62;63;64
```

Answer to Exercise E4.4. Creating a phenotype file from `.ped` data

The files are located in the directory `Exercise_ped2phe`:

```
$ cd ~/LinuxCourse/exercises_linux_course/Exercise_ped2phe
```

Different ways to show the first six columns of the first ten lines:

```
$ cut -f 1-6 -d " " chr.ped |head
$ head chr.ped | cut -f 1-6 -d " "
$ gawk '{print $1, $2, $3, $4, $5, $6}' chr.ped |head
$ head chr.ped | gawk '{print $1, $2, $3, $4, $5, $6}'
```

The commands that start with head are more efficient when working with files that have many lines. In that case first ten lines are selected from the file and subsequently piped to the other command. In the commands on the odd lines of the previous output, first the requested columns are selected for all lines and then only the first ten of those are printed on the screen.

For the phenotype file we only need columns two, five and six. The first two lines in the output below show different ways of creating the .phe file. The last line shows how to add the header as well.

```
$ cut -f 2,5,6 -d " " chr.ped > chr.phe
$ gawk '{print $2, $5, $6}' chr.ped > chr.phe
$ gawk 'BEGIN {print "id sex bt1"} {print $2, $5, $6}' chr.ped > chr.phe
```

And, finally, the one-liner for the complete .phe file:

```
$ gawk 'BEGIN {print "id sex bt1"} {if($5==2) sex=0; ↩
    if($5==1) sex=1; print $2, sex, $6}' chr.ped > chr.phe
```

For better legibility the command can be split into several lines. The > signs at the beginning of the lines are added by the shell and should not be typed by you. Here they don't indicate output redirection to a file (except, of course, the second > on line 8, which we do have to type ourselves).

```
1  $ gawk '
2  > BEGIN {print "id sex bt1"}
3  > {
4  > if($5==2) sex=0;
5  > if($5==1) sex=1;
6  > print $2, sex, $6
7  > }
8  > ' chr.ped > chr.phe
```

The BEGIN{} section is executed once before any other GAWK commands. It prints the header. Then the main body of the GAWK command starts (lines 3–7), it is repeated for each line in the .ped file. The body consists of two if-clauses that set the variable sex depending on the value of column five. Subsequently, in line 6 a print statement prints the data we want.

If this is a regularly recurring task it is of course better to save these lines in a GAWK script, say ped2phe.awk and run it repeatedly like this: gawk -f

```
$ gawk -f ped2phe.awk chr.ped > chr.phe
```

Answer to Exercise E4.5. Filtering output using gawk

The directory contains two files, one of 163 bytes and one of 710 KB:

```
$ ls -lh
total 716K
-rw-r----- 1 lennart lennart 163 2010-10-13 18:35 get_sign_snps.awk
-rw-r----- 1 lennart lennart 710K 2010-10-13 18:35 screen.1.out
```

Comparing the two GAWK one liners, we see that in the first case the word ”trait” can occur anywhere on the line, in the second we specify that the line should only be printed if the second field contains ”trait”.

These are the steps that schematically describe the filter. Because of the format of the data I made the assumption to use some tool that allows working with fields:

- The traits are listed on lines that begin with `Testing trait:`.
- SNP markers appear on lines that begin with `Testing marker:`.
- If white space (i.e. one or more spaces or tabs) is used to separate the fields then the lines that contain the information on *p* and *F* values that we are interested in have a 1 in the first field (i.e. the field named “Allele”).
- The *F* values are in field 6 and the *p*-values are found in field 7 (still assuming white space as field separator).
- However, if there is no *p*-value (as in line 15 of the output given in the exercise), then field 7 is not the *p*-value but a `(`.

The command `man gawk` or `gawk --help` shows that the `-f` option in combination with the script name is used to run a GAWK script.

```
$ gawk -f get_sign_snps.awk screen.1.out > qt_output
```

To find the number of lines in the output file use the word count program again:

```
$ wc -l qt_output
243 qt_output
```

Extracting the names of the traits becomes easy once you realise that they appear on a line with the word “trait” in it. So `grep` comes to the rescue:

```
$ grep trait qt_output
trait: NRUWE
trait: ERUWE
trait: ORUWE
trait: ARUWE
trait: CRUWE
```

Of course you don’t need to count the number of traits by hand! We’ve got (at least) two ways to do that:

```
$ grep trait qt_output | wc -l
5
$ grep -c trait qt_output
5
```

Answer to Exercise E5.1. A simple script

The script should look like this:

```
#! /bin/bash

echo "Hello World"
```

The executable bit is set like this: `chmod +x myscript.sh`. To check if the command did its job, run `ls -l` and look for the `x` in the permissions. It should be there for (at least) the user. In the example below the `x` permission is set for both the user and the group.

```
-rwxr-xr-- 1 lennart genepi 106 2010-11-02 10:28 myscript.sh
```

Answer to Exercise E5.2. Using variables

Variables are created like this:

```
variable_name="Some text"
```

Note that there are no spaces around the equal sign. To use a variable use the construction:

```
${variable_name}
```

So our script will look like this:

```
#! /bin/bash

# Create a variable
greeting="Good Morning"

echo "${greeting} World!"
```

Answer to Exercise E5.3. Using command line arguments in your script

The following script prints the first three arguments in reverse order.

```
#! /bin/bash
# This script prints the first three arguments
# in reverse order.

echo "${3} ${2} ${1}"
```

The modified script that takes the first argument as greeting looks like this:

```
1 #! /bin/bash
2
3 greeting=${1}
4
5 echo "${greeting} World!"
```

Of course the step in line 3 can be omitted, in which case ${1} goes directly into line 5.

The simplest way to use text that contains a space as one variable is to enclose it in quotes. If the above script would have been called `args.sh` then the this is the expected output:

```
$ ./args.sh Good evening
Good World!
$ ./args.sh "Good evening"
Good evening World!
```

Answer to Exercise E5.4. For-loops

There are two points to note here. First, the semi-colon before `do`, it's easily forgotten. Second, it's easiest to construct the file name in a variable and use that together with the fictitious `analyse` command.

```
1 #! /bin/bash
2 for number in {22..1..-1}; do
3   filename="chr${number}.dat"
4   analyse ${filename}
5 done
```

Of course lines 3 and 4 can be written as one as well:

```
analyse "chr${number}.dat"
```

Adding leading zeroes is easy:

```
for number in {22..01..-1}; do
```

Answer to Exercise E5.5. parallel for-loops

In order to start a process in the background you need to add an `&` after the command. So an example script that would start each job in parallel would look like this:

```
#! /bin/bash
# This script runs all tasks at the same time

for i in {01..22}; do
 my_analysis_script.sh ${i} &
```

```
done
```

As the question already hinted, this is not an ideal way to run things in parallel of you have more tasks than CPUs. In the ideal case you want each CPU to be busy with only one task. so if you have 10 CPUs you need to split the tasks up in surch a way that first the first 10 are run, then the second 10 and finally the remaining two. If you know that you always have 22 tasks you could simply write three for-loops with a `wait` command after each loop. The `wait` command was not discussed previously. It waits for all jobs that were started previously in the script to end before continuing with the next command.

Answer to Exercise E5.6. if-clauses and tests

First we have to check if a command line argument was given. Command line arguments are stored in `${1}` etc (cf. Exercise E5.3). From Table 5.1 we see that we can use `-n` to test if this variable is an empty string.

```
#! /bin/bash
if [ -n "${1}" ]; then
   echo "The command line argument was ${1}"
else
   echo "Goodbye!"
fi
```

Checking whether a directory exists can be done with the `-d` test. To negate that test (because if it is not there we must take action) use `!`.

```
#! /bin/bash

# First create a variable that contains today's date in
# an acceptable form for a directory name.
today=$(date +%F)

if [ ! -d ${today} ]; then
   echo "A directory with today's date does not exist."
   echo "Creating it..."
   mkdir ${today}
fi
```

Answer to Exercise E6.1. Working with Sun Grid Engine

The test script that will be submitted to the queue will look like this:

```
1 #! /bin/bash
2 for i in $(seq 1 5); do
3   whoami
4   date +%T
5   sleep 10
6 done
7 date
8 echo "--------------------"
```

If you want the user name and the time to appear on a single line, lines 3 and 4 can be replaced with

```
echo "$(whoami): $(data +%T)"
```

Note that `$()` needs to be used, otherwise it would print the literal commands instead of their output.

If the script is called `simplejob.sh`, then it can be submitted with this command:

```
$ qsub -cwd simplejob.sh
```

To check the status of your jobs use the `qstat` command. The usual reason for a job not being run immediately is the fact that the maximum number of active slots has filled up (at present a maximum of 7 jobs is allowed to run at the same time). Use

```
$ qstat -f -u \*
```

to show all jobs in all queues for all users. This will also give you an indication of the priority of your job compared to others. The job at the top of the waiting list will be run at the next time a slot becomes available. SGE is configured for fair scheduling, this means that it tries to allot each user the same amount of computation time. If, for example, you have several jobs running and several more waiting to be run, than someone else's job will be scheduled with a higher priority (assuming this person didn't over use his fair share of computation time). This makes sure that one person with 100 jobs will not block others from running their own analyses.

When the script has finished, check the files that end in `.e5678` and `.o5678`, where `5678` is to be replaced with the job ID of your own job.

Answer to Exercise E6.2. SGE, R and command line arguments

The R script looks a lot like the example. Be sure to convert the arguments to integers before using them. The if-clause in lines 4 – 7 gives an error message if the user forgets one or more arguments. It is good practice to build these kind of sanity checks in your code. They help you if you run the same script next year and have forgotten how it exactly works.

```
1  cat("Start of the program\n")
2  
3  arguments <- commandArgs(TRUE)
4  if (length(arguments) < 2) {
5    cat("Error: please give two command line arguments\n")
6    q()
7  
8  
9  start <- as.integer(arguments[1])
10 end  <- as.integer(arguments[2])
11 
12 for (i in start:end) {
13   cat(i, "^2 = ", i^2, "\n", sep="")
14 }
15 
16 cat("End of program\n")
```

Running this program gives the following results:

```
$ R --slave --vanilla -q -f R_loop_script.R --args 1 5
Start of the program
1^2 = 1
2^2 = 4
3^2 = 9
```

```
4^2 = 16
5^2 = 25
End of program
```

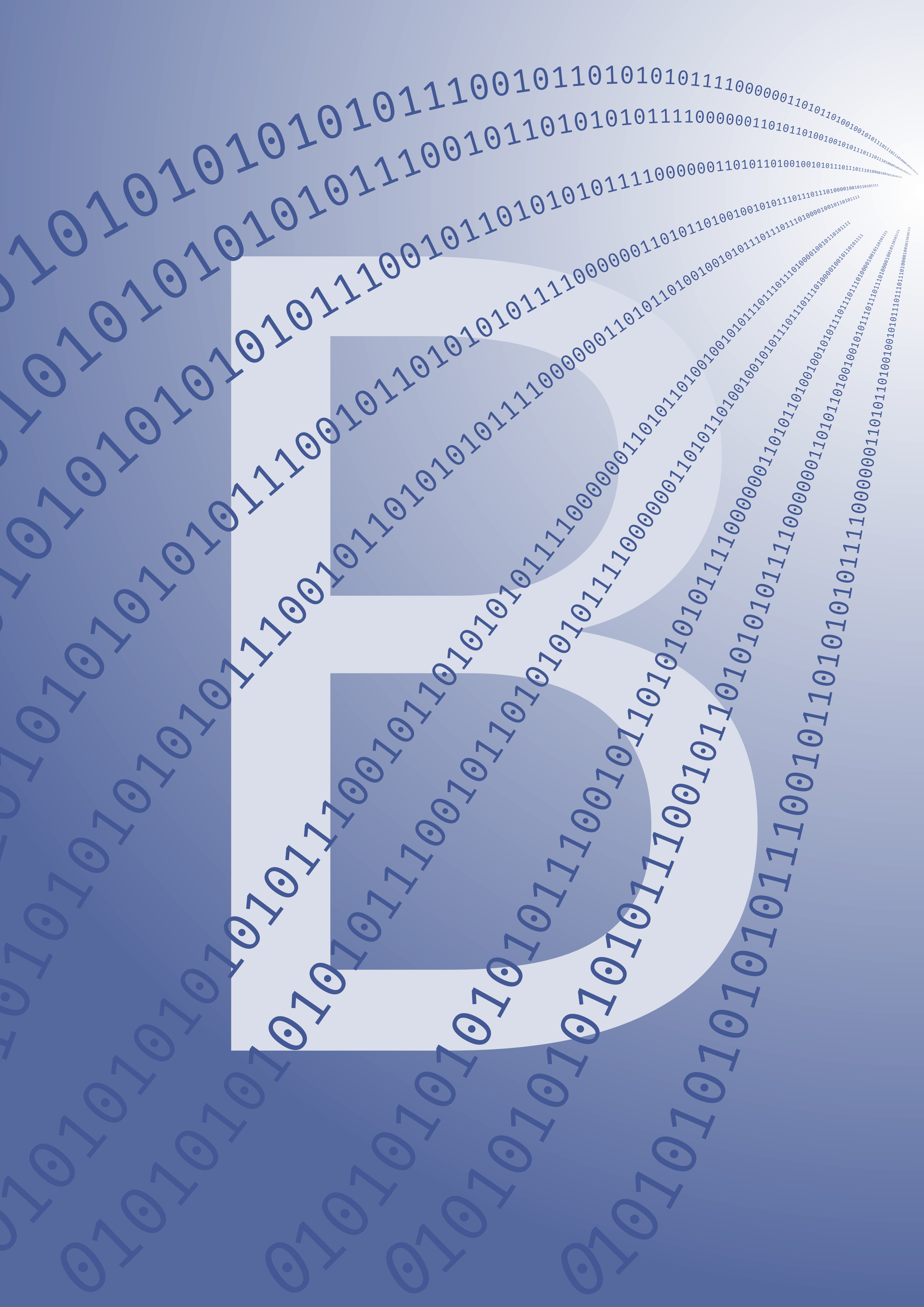
This is the modified job script:

```
 1 #! /bin/bash
 2 # This is an example of a job submission script
 3 
 4 # The following are options for qsub. Don't
 5 # remove the # in front of them.
 6 #$ -S /bin/bash
 7 #$ -j y
 8 #$ -cwd
 9 
10 # Here comes the rest of your script that
11 # actually does something. In this case it runs R.
12 message="This script needs two arguments"
13 
14 if [ -z ${1} ]; then
15    echo $message
16    exit
17 fi
18 if [ -z ${2} ]; then
19    echo $message
20    exit
21 fi
22 
23 R --slave --vanilla -q -f R_loop_script.R --args ${1} ${2}
```

In line 12 a variable is created with the error message. Lines 14 – 21 test the presence of the arguments, which are subsequently sent to the R script in line 23.

Submission to the queue goes like this:

```
$ qsub jobscript.sh 1 10
```


B

Appendix B

Reference Card of Basic Linux Commands

Appendix B Reference Card of Basic Linux Commands

The next two pages form a reference card of basic Linux commands that can be printed for easy reference.

Reference Card of Basic Linux Commands

©2010–2019 Lennart C. Karssen, `l.c.karssen@polyomica.com`
Based on the vi Reference Card by Donald J. Bindner.

Logging out

	`exit`
	`logout`
	`Ctrl-d`

Getting help

manual page on *command*	`man command`
shows basic help and options for *command*	`command --help`

Keyboard shortcuts

The most important key is `TAB`. It is used to complete commands and file names. If pressed twice it will list all available completions.

cancel current command	`Ctrl-c`
go to beginning of line	`Ctrl-a`
go to end of line	`Ctrl-e`
search history for a recent command	`Ctrl-r`
undo your last change on the command line[1]	`Ctrl-/`
delete to end of line and copy to buffer (cut)	`Ctrl-k`
delete to beginning of line and copy to buffer (cut)	`Ctrl-u`
paste (or: yank) from buffer	`Ctrl-y`
suspend current job	`Ctrl-z`
insert last word from previous command	`Alt-.`

File and directory management

shortcut for user's home directory (`/home/user/`)	`~`
the current directory	`.`
the parent directory	`..`
print present working directory	`pwd`
change directory to *mydir*	`cd mydir`
change to your home directory	`cd`

Listing files

list files and directories	`ls`
list files and directories with a `/` after each directory	`ls -p`
detailed (long) list with file sizes in human readable format	`ls -lh`
list hidden files and directories as well	`ls -a`
list files in reverse order of creation time (i.e. newest last)	`ls -rt`

Copy, (re-) move, rename

create a directory *mydir*	`mkdir mydir`
remove an empty directory *mydir*	`rmdir mydir`
copy[2] *file1* to *file2*	`cp file1 file2`
copy a directory recursively to *dir2*	`cp -r dir1 dir2`
copy a file to a directory on a remote host	`scp myfile user@host:~/dir/`
copy a directory recursively from a remote host to the current directory	`scp -r user@host:~/mydir .`
a smarter way of copying a directory to a remote host	`rsync -azP dir1 user@ost:~/dir2/`
remove[3] a file	`rm myfile`
remove a directory recursively (including all its files)	`rm -r mydir`
move a file to another directory	`mv myfile mydir`
rename a file	`mv file1 file2`

Finding files

quick way of finding a *myfile* (not always up to date)	`locate myfile`
idem, but file name case insensitive	`locate -i file`
find file in a certain directory *startdir*	`find startdir -name "myfile"`
idem, but case insensitive	`find startdir -iname "myfile"`

File and directory permissions

File and directory permissions can be set to `read` (`r`), `write` (`w`) and `executable` (`x`) for the following: owner, group and others. For a directory `x` means it is accessible. Use `ls -l` to show these permissions.

safe permissions for a file	`rw-r-----`
safe permissions for a directory	`rwxr-x---`
set executable bit on *myscript.sh* (owner and group)	`chmod +x myscript.sh`
make file readable to all users	`chmod a+r myfile`
recursively remove `read` perm. from *mydir* for 'others'	`chmod -R o-r mydir`
remove write permission for 'group' from *myfile*	`chmod g-w myfile`

Wildcards and shell expansion

Wildcards are used to select more than one file (for example for copying or deletion). Be careful with them!

select one character	`?`
select one or more characters	`*`
select one character from a range[4]	`[]`
select from list of characters	`{}`

For example:

list all files ending in `.pdf` in this directory	`ls *.pdf`
copy all `.R` files to another directory	`cp *.R mydir`
find all files with `chr2` in their name in `~`	`find ~ -iname "*chr2*"`
move `file_1.txt` – `file_9.txt` to *mydir* (but also `file_a.txt`, `file_B.txt`, etc.)	`mv file?.txt mydir`
move `file_1.txt` – `file_5.txt` to *mydir* (but not `file_a.txt`, `file_6.txt`, etc.)	`mv file_[1-5].txt mydir`

show info of all files with `.dose` or `.prob` extensions	`ls -lh ←`
	`*.{dose,prob}`

Process management

display processes in your current terminal	`ps`
display all processes by all users	`ps -ef`
display all processes by user *user*	`ps -fu user`
display all running processes dynamically	`htop`
display all running processes dynamically	`top`
kill process id *pid*	`kill pid`
kill all processes named *proc*	`killall proc`
suspend current job	`Ctrl-z`
resume a suspended job in the background	`bg`
list jobs	`jobs`
bring the most recent job to foreground	`fg`
bring job *n* to the foreground	`fg n`
immediately start *command* in the background	`command &`

Display contents of a file

show full contents of *myfile*	`cat myfile`
show contents screen by screen	`more myfile`
idem, but with search and scrolling	`less myfile`
Use `q` to exit `more` and `less`.	
show first 10 lines of *myfile*	`head myfile`
show first 4 lines of *myfile*	`head -n 4 myfile`
show last 10 lines of *myfile*	`tail myfile`
show last 15 lines of *myfile*	`tail -n 15 myfile`
show last 10 lines of *myfile* and keep following it as it grows	`tail -f myfile`

System info

display users that are logged in	`w`
get information about *user*	`finger user`
show how long the server has been up and running	`uptime`
show your total disk usage (and how much space is left)	`quota -s`
show disk usage in a certain directory *dir*	`du -s dir`
idem, but in human readable format	`du -sh dir`

Compression and archiving

compress a single file *myfile* (original file is removed)	`gzip myfile`
decompress a single file (`.gz` file is removed)	`gunzip myfile.gz`
create a compressed archive of *mydir*	`tar -czf myfile.tar.gz mydir`
extract all files from a `.tar.gz` file to `.`	`tar -xzf myfile.tar.gz`
idem, but show the file names that are extracted	`tar -xzvf myfile.tar.gz`
create a compressed archive of *mydir*	`tar -cjf myfile.tar.bz2 mydir`
extract all files from a `.tar.bz2` file to `.`	`tar -xjf myfile.tar.bz2`

Text manipulation

replace all occurrences of *old* with *new* in *myfile*	`sed -i "s/old/new/g" myfile`
print line 4 from *myfile*	`sed -n "4p" myfile`
delete line 25 from *myfile*	`sed -i '25d' myfile`
delete lines 2 to 5 from *myfile*	`sed -i '2,5d' myfile`
print column 3 from *myfile* using commas as column delimiter	`cut -f 3 -d "," myfile`
print columns 1 and 3 to 7 from *myfile* using tab as column delimiter	`cut -f 1,3-7 myfile`
print 1st, 2nd and last column from *myfile* (columns separated by white space)	`gawk '{print $1, $2, $NF}' myfile`
idem, but columns are separated by a comma	`gawk -F"," '{print $1, $2, $NF}' myfile`
print all columns except the 3rd and 5th from *myfile*	`gawk '{$3=$5=""; print $0}' myfile`
print line from *myfile* if first column contains *text*	`gawk '$1 ~ text {print $0}' myfile`
change column 3 to NA in *myfile*	`gawk '{$3="NA"; print $0}' myfile`
idem, but don't change the header line	`gawk '{if (NR!=1){$3="NA"}; print $0}' myfile`
select the field on row 2, column 3 from *myfile*	`gawk '{if (NR==2)print $3}' myfile`
sort lines in a file (by default uses column 1, separated by space)	`sort myfile`
idem, but assume data is numeric, not character	`sort -n myfile`
idem, but assume human readable data (1k, 2M, etc.)	`sort -h myfile`
idem, but assume numbers are in scientific notation (e.g. `1.2e-3`)	`sort -g myfile`

[1] This only undoes changes on the command line itself (before you hit the Enter key), not the results of the commands that you type.

[2] Note: `cp` overwrites the destination file (if it exists) without notice. Use the `-i` option to get notified.

[3] `rm` does not ask for confirmation. Use the `-i` option if you want to be asked for each file.

[4] Note, only one character is substituted here, so `[0-9]` will work, but `[1-22]` will not.

sort myfile based on column 2	`sort -k2 myfile`
idem, and use a comma to separate the columns	`sort -t"," -k2 myfile`
remove adjacent (!) duplicate lines from *myfile*	`uniq myfile`

Output redirection

send screen output of *command* to *myfile* (which will be overwritten)	`command > myfile`
idem, but append to instead of overwrite *myfile*	`command >> myfile`
use output of *command1* as input for *command2*	`command1 | command2`

Differences between files

show differences between two text files	`diff file1 file2`
idem, but with more context	`diff -u file1 file2`
for binary files: compare md5sums	`md5sum file1 file2`

Searching for text

print lines from *myfile* containing *pattern*	`grep pattern myfile`
idem, but faster (assumes search text is fixed)	`grep -F text myfile`
idem, but `pattern` is treated as a word (e.g. with surrounding white space)	`grep -w pattern myfile`
idem, but *pattern* is treated case insensitive	`grep -i pattern myfile`
idem, and prefix with line number	`grep -n pattern myfile`
idem, but get search patterns from file `search.txt`	`grep -f search.txt myfile`
idem, but *file1* contains a list of *patterns*	`grep -f file1 myfile`
print lines containing *pattern* and 3 lines after	`grep -A 3 pattern myfile`
print lines containing *pattern* and 4 lines before	`grep -B 4 pattern` ↩ `myfile`
print lines containing *pattern* and 5 lines context	`grep -C 5 pattern` ↩ `myfile`
print lines without *pattern* in *myfile*	`grep -v pattern myfile`
search recursively for *pattern* in *mydir*	`grep -r pattern mydir`
count the number of lines on which `pattern` occurs	`grep -c pattern myfile`

Miscellaneous

get more information about the contents of a file	`file myfile`
generate a sequence of numbers from 1 to 22	`{1..22}`
generate a sequence of numbers from 1 to 22	`seq 1 22`
idem, but with steps of 4	`{1..22..4}`
idem, but with steps of 4	`seq 1 4 22`
idem, but counting down with a step of -1	`{22..1..-1}`
idem, but counting down with a step of -1	`seq 22 -1 1`
show current date and time	`date`
show only today's date	`date +%F`
show only current time	`date +%T`
convert date to other format	`date -d "01 March 96" +%F`
convert time in other time zone to time in server's time zone	`date -d "11:00 pm PDT" +%T`
change your password	`passwd`
create an alias for a command[5]	`alias short='long command'`
for example:	`alias lsl='ls -lhart'`

Bash scripting

Bash scripts should be made executable	`chmod +x myscript.sh`
All bash scripts begin with	`#!/bin/bash`
Exit script when a command gives error	`set -e`
Exit script when a variable is undefined	`set -u`
Exit script if part of a pipeline errors	`set -o pipefail`
fill variable *myvar* with `some text`	`myvar="some text"`
fill variable *myvar* with the output of *command*	`myvar=$(command)`
use the variable *myvar*	`${myvar}`

Example of a for-loop: run *myscript.sh* for each chromosome

```
for chr in {1..22}; do
   myscript.sh ${chr}
done
```

Example of a Bash if-clause that compares two strings:

```
if [ "${var}" = "yes" ]; then
   echo "The variable var is equal to yes"
else
   echo "var wasn't yes"
fi
```

Other string comparisons are:

strings: equal	`=`
strings: not equal	`!=`
string `s1` is not empty	`-n s1`
string `s1` is empty	`-z s1`
numbers: equal	`-eq`
numbers: not equal	`-ne`
numbers: less than	`-lt`
numbers: greater than	`-gt`

Testing files and directories in an if-clause:

Check for directory existence	`-d directory`
Check for file existence	`-e file`
Check for regular file existence not a directory	`-f file`
Check if file is a readable	`-r file`
Check if file is writable	`-w file`
Check if file is executable	`-x file`

For example (the `!` in the second line is a logical NOT, so the test is whether `somescript.sh` is not executable):

```
filename=some_script.sh
if [ ! -x ${filename} ]; then
   echo "${filename} is not executable. Fixing..."
   chmod +x ${filename}
fi
```

Checking the number of command line arguments in a script:

```
if [ ${#} -lt 3 ]; then
   echo "The number of arguments should be at least 3"
   exit
fi
```

Cycling through all command line arguments in a script:

```
for arg in ${*}; do
   echo "Argument was: ${arg}"
done
```

Initialise array one element at a time	`arr[index]=value`
Initialise array	`arr=(elm0 elm1 elm2)`
Accessing array element at `index` (first element has index `0`!)	`${arr[index]}`
Accessing all elements	`${arr[@]}`
idem	`${arr[*]}`
Get array size/length (nr. of elements)	`${#arr[@]}`
Get length of array element 2	`${#arr[2]}`
Appending elements to an array	`arr=(${arr[@]} new1 new2)`

Using an array in a loop:

```
for i in ${arr[@]}; do
   echo "Element ${i} = ${arr[${i}]}"
done
```

Since arrays are space separated looping over files is easy:

```
for fl in *.csv; do
   do_something
done
```

The SGE batch queue system

list your jobs in the queue(s)	`qstat`
list all jobs by all users in all queues	`qstat -f -u \*`
get info on a waiting or running job with JobID 1234	`qstat -j 1234`
get info on a finished job with jobID 1234	`qacct -j 1234`
submit a job script to the queues	`qsub myscript.sh`
submit a binary job to the queue system	`qsub -cwd -b y myprogram`
submit a job script and give it a name	`qsub -N jobname myscript.sh`
submit a job that should until a job with name `myjob` has finished	`qsub -hold_jid myjob myscript.sh`
delete job with jobID 1234 from the queues	`qdel 1234`

The `qsub` command is used to send jobs to the queue. Without further options it will look for your script in your home directory and also put its output there. These are the most important options for the `qsub` command:

use current working directory for input and output files	`-cwd`
join the normal output and the errors in one file	`-j y`
the submitted command is binary, not a script	`-b y`
send e-mail at beginning and end of a job	`-m b e`
set e-mail address for the `-m` option	`-M user@example.com`
array job for 22 subjobs (use variable `$SGE_TASK_ID` in your script)	`-t 1-22`
array job with different step size, e.g. 5	`-t 100-200:5`
array job, but start max 2 jobs at the same time	`-t 1-300 -tc 2`

When submitting a shell script the aforementioned options can be set in the script so you don't have to retype them. Simply put them in the script on a line that starts with `#$`. For example:

```
#!/bin/bash
# This is an example of a job submission script

# The following are options for qsub.
#$ -S /bin/bash
#$ -j y
#$ -cwd

# Here comes the rest of your script that
# actually does something.
```

[5] Save aliases in your `~/.bash_profile` file so they will be loaded next time you log in.

Appendix C

List of acronyms

BSD Berkeley Software Distribution, originally a Unix-like OS, currently used to denote the whole family including its descendants.

CLI Command Line Interface

CPU Central Processing Unit, a.k.a. the processor of your computer.

ErasmusMC Erasmus University Medical Centre, the place where the foundations for this book were laid.

GIYF Google Is Your Friend

GNU GNU's Not Unix, a recursive acronym used by the GNU free software project (`http://www.gnu.org`).

GPL GNU Public License, a so-called copyleft licence that gives the user liberal rights with respect to the use and distribution of software.

GUI Graphical User Interface

NIHES Netherlands Institute for Health Sciences, `http://www.nihes.nl`.

OS Operating System, the main pieces of software that make a computer run.

pwd present working directory

SGE Sun Grid Engine, a batch queue system used the schedule compute-intensive jobs on one or more servers.

TIMTOWTDI There Is More Than One Way To Do It

Bibliography

[1] The Linux Foundation. *The Story of Linux: Commemorating 20 Years of the Linux Operating System*. Aug. 2011. URL: http://youtu.be/5ocq6_3-nEw.

[2] Linus Torvalds. *Gewoon Voor De Fun: Het genie achter Linux*. ISBN: 90-6112-831-5. Karakter Uitgevers BV, 2001.

[3] L. Torvalds and D. Diamond. *Just for Fun: The Story of an Accidental Revolutionary*. HarperBusiness, 2002. ISBN: 978-0-066-62073-2. URL: http://books.google.com/books?id=6zSWd8Ou8BAC.

[4] W3Techs. *Usage statistics and market share of Unix for websites*. 2012. URL: http://w3techs.com/technologies/details/os-unix/all/all.

[5] Top 500 Supercomputer Sites. *Top 500 supercomputing operating system share*. June 2012. URL: http://i.top500.org/stats/list/38/os.

[6] Yurii S. Aulchenko, Maksim V. Struchalin, and Cornelia M. van Duijn. "ProbABEL package for genome-wide association analysis of imputed data." eng. In: *BMC Bioinformatics* 11 (2010), p. 134. DOI: 10.1186/1471-2105-11-134. URL: http://dx.doi.org/10.1186/1471-2105-11-134.

[7] Eric Steven Raymond. *The Art of Unix Programming*. 2003. URL: http://www.faqs.org/docs/artu/index.html.

[8] Keith Bradnam and Ian Korf. *UNIX and Perl to the rescue!: A Field Guide for the Life Sciences (and Other Data-rich Pursuits)*. ISBN: 9781107000681. Cambridge University Press, 2012.

[9] John Loeliger and Matthew McCullough. *Version Control with Git*. Ed. by Andy Oram. 2nd. ISBN: 978-1-449-31638-9. O'Reilly, 2012. URL: http://shop.oreilly.com/product/0636920022862.do.

[10] Arnold Robbins and Nelson H.F. Beebe. *Classic Shell Scripting*. ISBN: 978-0-596-00595-5. O'Reilly, 2005. URL: http://shop.oreilly.com/product/9780596005955.do.

[11] Carl Albing, JP Vossen, and Cameron Newham. *Bash Cookbook: Solutions and Examples for Bash Users*. 1st ed. O'Reilly, 2007. ISBN: 0596526784.

[12] V. Anton Spraul. *Think Like a Programmer: An Introduction to Creative Problem Solving*. ISBN: 9781593274245. No Starch Press, 2012. ISBN: 1593274246.

[13] Roderick W. Smith. *LPIC-1: Linux Professional Institute Certification Study Guide: (Exams 101 and 102)*. ISBN: 978-0-470-40483. Sybex, 2009.

[14] Debra Cameron et al. *Learning GNU Emacs*. third. ISBN: 978-0-596-0064-88. O'Reilly, 2004.

[15] Mickey Petersen. *Mastering Emacs*. Sept. 26, 2016. URL: https://masteringemacs.org/about.

Index

Colophon

This book was typeset with XƎLaTeX using the *Fira Sans Book* font v4.1 by Mozilla (`http://www.carrois.com/fira-4-1/`). The *Fira Mono* font v3.2 was used for monospaced text. BibLaTeX was used to generate the bibliography.