Introduction to Computational Physics for Undergraduates

Online at: https://doi.org/10.1088/978-1-6817-4896-2

Introduction to Computational Physics for Undergraduates

Omair Zubairi
Wentworth Institute of Technology

Fridolin Weber
San Diego State University and University of California at San Diego

Morgan & Claypool Publishers

ISBN 978-1-6817-4896-2 (ebook)
ISBN 978-1-6817-4893-1 (print)
ISBN 978-1-6817-4894-8 (mobi)

DOI 10.1088/978-1-6817-4896-2

Version: 20180301

IOP Concise Physics
ISSN 2053-2571 (online)
ISSN 2054-7307 (print)

A Morgan & Claypool publication as part of IOP Concise Physics
Published by Morgan & Claypool Publishers, 1210 Fifth Avenue, Suite 250, San Rafael, CA, 94901, USA

IOP Publishing, Temple Circus, Temple Way, Bristol BS1 6HG, UK

Contents

Preface

This introductory textbook on computational physics intended for undergraduates at the sophomore or junior level who have taken the introductory freshman series of physics courses to include: introductory classical mechanics, electricity and magnetism, and modern physics. A good understanding of multivariable calculus and linear algebra is highly encouraged. This text provides an introduction to programming languages such as FORTRAN 90/95 and covers numerical techniques such as differentiation, integration, root finding, and data fitting. The textbook also entails the use of the Linux/Unix operating system, text editors, and python for plotting data.

This textbook will allow the reader to become a proficient user of the Linux/Unix operating system. The reader will able to write, compile, and debug computer code in the FORTRAN programming language. The reader will also be able to apply computational techniques such as iterative processes, logical conditions, and memory allocation in addition to applying numerical methods to solve problems involving differentiation, integration, matrix theory, and root finding. The reader will able to use the contents of this text and apply them to a variety of science and engineering applications.

Acknowledgments

F Weber thanks the National Science Foundation (USA) for supporting his efforts to create opportunities for undergrad students to take part in a range of research and learning activities. O Zubairi acknowledges Wentworth Institute of Technology's EPIC mini grant program which has supported his endeavors to create computational resources for student advancement in research in and out of the classroom.

Author biographies

Omair Zubairi

Omair Zubairi received his BSc and MSc in Physics from San Diego State University. He obtained his PhD in Computational Science from Claremont Graduate University and San Diego State University where he primarily worked on compact star physics. Omair is currently an Assistant Professor of Physics at Wentworth Institute of Technology. His other research interests include general relativity, numerical astrophysics and computational methods and techniques.

Omair is a dedicated educator in physics and computational science. He has taught students from all backgrounds in many areas of physics from the introductory sequence to upper division courses where he incorporates numerical methods and computational techniques into each course. 'By allowing students to see and apply numerical simulations to various topics covered in lectures, they are able to gain invaluable insight into the problem at hand.'

Fridolin Weber

Fridolin Weber is a Distinguished Professor of Physics at San Diego State University and a Research Scientist at the University of California at San Diego. He is interested in nuclear and particle processes that occur in extreme astrophysical systems such as neutron stars and supernovae. Other interests include the application of quantum many-body theory to nuclear matter and dense quark matter, relativistic astrophysics, and Einstein's theory of general relativity. Dr Weber has a PhD in theoretical nuclear physics and a PhD in theoretical astrophysics, both from the Ludwig Maximilian University of Munich, Germany. He has published two books, is the author or co-author of almost 200 publications, and has given around 300 talks at conferences and physics schools.

Chapter 1

The Linux/Unix operating system

1.1 Introduction

The main purpose of this introduction is to make you familiar with the interactive use of Unix/Linux for day-to-day organizational and programming tasks. Unix/Linux is an operating system (OS) which we can loosely define as a collection of programs (often called processes) which manage the resources of a computer for one or more users. These resources include the CPU, network facilities, terminal windows, file systems, disk drives and other mass-storage devices, printers, and many more. During this course, the most common way you will use Unix/Linux is through a command-line interface; you will type commands to create and manipulate files and directories, start up applications such as text editors or plotting packages, and compile and run Fortran programs,

When you type commands in Unix/Linux, you are actually interacting with the OS through a special program called a shell which provides a user-friendly command-line interface. These command-line interfaces provide powerful environments for software development and system maintenance. Although shells have many commands in common, each type has unique features. Over time, individual programmers come to prefer one type of shell over another. We recommend that you use the 'C-shell' (`csh`), the 'tC-shell' (`tcsh`), or the 'bash shell' (`bash`) for interactive use.

All the Unix/Linux commands described below are bash shell features. The bash shell offers command-history recall and editing via the 'arrow' keys (as well as 'delete' and 'backspace'). After you have typed a few commands, hit the 'up arrow' key a few times and note how you scroll back through the commands you have previously issued. In the following, we shall assume that you have at least one active shell on each system in which to type Unix/Linux commands, and we will often refer to a window in which a shell is executing commands across the book. as the terminal. Popular terminal windows on Unix/Linux machines are iTerm, aterm, and xterm. An example of the latter is shown in figure 1.1. Henceforth, commands typed

```
> 
```

Figure 1.1. The X-terminal window on a machine running Ubuntu Linux.

to the shell at the shell prompt (denoted by '>') are shown in red typewriter fonts, while the shell response is shown in blue typewriter fonts. Here is an example:

```
> pwd                                    ↩
/home/student
> whoami                                 ↩
student
> ps -p $$ -o comm=""                    ↩
-bash
```

1.2 Files and directories

There are essentially three types of files in Unix/Linux. These are

- regular files, such as plain text files, source code files, executables, postscript files;
- directory files, which contain other files and/or directories; and
- special files, such as block files, character device files, named pine files, symbolic link files, and socket files.

1.2.1 Pathnames and working directories

All Unix/Linux file systems are rooted in the special directory called '/'. All files within the file system have absolute pathnames which begin with '/' and which describe the path down the file tree to the file in question.

Thus

```
/home/student/sample.txt
```

refers to a file named `sample.txt` which resides in a directory with absolute pathname

```
/home/student/
```

which itself lives in directory

```
/home
```

which is contained in the root directory, `/`. In addition to specifying the absolute pathname, files may be uniquely specified using relative pathnames. The shell maintains a notion of your current location in the directory hierarchy, known appropriately enough, as the working directory. The name of the working directory may be printed using the `pwd` command:

```
> pwd                                                    ↩
/home/student/
```

If you refer to a filename such as

```
file.txt
```

or a pathname such as

```
dir1/dir2/file.txt
```

so that the reference does not begin with a '/', the reference is identical to an absolute pathname constructed by prepending the working directory followed by a '/' to the relative reference. Thus, assuming that your working directory is

```
/home/student/txt
```

the two previous relative pathnames are identical to the absolute pathnames

```
/home/student/txt/file.txt
/home/student/txt/dir1/dir2/file.txt
```

Note that although these files have the same filename `file.txt`, they have different absolute pathnames and hence are different from each other.

Each user of a Unix/Linux system typically has a single directory called his/her home directory which serves as the base of his/her personal files. The command `cd` (change directory) with no arguments will always take you to your home directory. On your Linux machine you may see something like this:

```
> cd                                    ↩
> pwd                                   ↩
/home/student
```

When using the C-shell, you may refer to your home directory using a tilde ('~'). Thus, assuming the home directory is `/home/student`, then

```
> cd ~
```

followed by

```
> cd dir1/dir2
```

is identical to

```
> cd /home/student/dir1/dir2
```

Unix/Linux uses a single period ('`.`') and two periods ('`..`') to refer to the working directory and the parent of the working directory, respectively:

```
> cd ~/student/homework1                ↩
> pwd                                   ↩
/home/student/homework1
> cd ..                                 ↩
> pwd                                   ↩
/home/student
> cd .                                  ↩
> pwd                                   ↩
/home/student
```

Note that

```
> cd .
```

does nothing—the working directory remains the same. However, the '/' notation is often used when copying or moving files into the working directory. See below for more details.

1.2.2 Filenames

There are relatively few restrictions on filenames in Unix/Linux. On most systems (including Linux machines), the length of a filename cannot exceed 255 characters. Any character except the forward slash ('/') and 'null' may be used. However, you should avoid using characters which are special to the shell, such as '(', ')', '*', '?', '$', '!' as well as blanks (spaces). In other words, using upper- and lower-case letters, numbers and a set of symbols, as shown below, is highly recommended,

```
a – z, A – Z, 0 – 9, _, ., –
```

which includes underscores, periods, and dashes. As is the case for other operating systems, the period is often used to separate the 'body' of a filename from an 'extension'. Examples are shown in table 1.1, where the full filenames are listed in the left column and the extensions in the right column. Note that unlike some other operating systems, extensions are not required, and are not restricted to some fixed length. Several standard Unix/Linux filename extensions are shown in table 1.2. The underscore and dash sign are often used to create more human readable filenames such as This_is_better, which is better readable than a file named Thisisnotsogood.

If one accidentally creates a filename containing characters which are special to the shell, such as '*' or '?', it is best to rename or move (mv) this file. This is done by enclosing the file's name in single forward quotes to prevent shell evaluation. Below we show an example for a text file which contains an asterisk:

```
> mv 'bad_file*_name.txt'   good_file_name.txt   ↩
```

The mv command renames the file specified on the command line. The single quotes must be forward quotes as backward quotes have a completely different meaning to the shell.

Table 1.1. Examples of file extensions.

Full file name	Extension
program.f	.f
program.f90	.f90
paper.tex	.tex
document.txt	.txt

Table 1.2. Overview of standard Unix/Linux filename extensions.

File extension	Usage
`.c`	C language source code
`.cpp`	C++ language source code
`.f`	Fortran 77 language source code
`.f90`	Fortran 90 language source code
`.o`	Object code generated by a compiler
`.pl`	Perl language source code
`.ps`	PostScript language source
`.tex`	TEX or LaTeX document
`.dvi`	Device independent output file
`.gif`	Graphic Interchange Format (GIF) graphics file
`.jpg`	Joint Photographic Experts Group (JPE) graphics file
`.tar`	Archive file created with `tar`
`.Z`	Compressed file created with `compress`
`.tgz`	Compressed (gzipped) archive file created with `tar`
`.a`	Library archive file created with `ar`

1.3 Overview of Unix/Linux commands

Beginning Unix/Linux users are often overwhelmed by the number of commands they must learn in order to perform tasks. To assist such users, we discuss in this chapter the most commonly used Unix/Linux commands, which will allow users to perform many essential operations on Unix/Linux machines. An overview of the most important commands is provided in table 1.3. The general structure of Unix/Linux commands is schematically given by

```
command_name [options] [arguments]
```

where the square brackets may contain optional parameters. Options to Unix/Linux commands are frequently single alphanumeric characters preceded by a minus sign as in this example:

```
> ls -l          ↩
> cp -R ...      ↩
> man -k ...     ↩
```

where the ellipses stand for directory names or commands which have been omitted. They are typically provided as arguments to shell commands, which do not start

Table 1.3. Summary of essential Unix/Linux commands.

Topic	Command	Examples
List filenames	ls	* ls -l .c, ls -a, ls -F, ls -alF
Move files or directories	mv	mv temp.txt newfile.txt mv temp.txt ../new/list.txt
Copy files or directories	cp	cp temp.txt newfile.txt cp temp.txt ../new/list.txt
Remove a file or directory	rm	rm temp.txt rm -i temp.txt rm -rf directory
Look the MANual pages for a command	man	man rm
The '-k' option searches man pages for keyword		man -k xterm
Make a directory	mkdir	mkdir newdir
Remove a directory	rmdir	rmdir newdir
Change directory	cd	cd texdir
Print working directory	pwd	pwd
Send file to a printer	lpr, lp	lp -Pprintername filename
List content of file	cat	cat file1
	more	more file1
	less	less file1
Print string or variable	echo	echo $USER echo "hello, world"
To see list of recent commands	history	history
Set protection of a file	chmod	chmod 755 file
Set owner of a file	chown	chown smith file
Make a link (alias) to a path	ln	ln -s ~/classes/phys-317 phys-317
Find out disk quota	quota	quota -v
Find out disk usage	du	du
Create archive file tarfile.tar from list of files (can be a directory)	tar -cvf	tar -cvf tarfile.tar list
Create gzipped archive file from list of files	tar -czvf	tar -czvf tarfile.tar.tgz list
Extracts files from archive file tarfile. tar	tar -xvf	tar -xvf tarfile.tar
Extract files from a gzipped tarfile	tar -xzvf	tar -xzvf tarfile.tar.gz
Zip filename (can be tar file) into compressed	gzip	gzip filename
file filename.gz		
Unzip filename from filename.gz	gunzip	gunzip filename.gz
Another file compression	bzip2	bzip2 filename
Decompressing files	bunzip2	bunzip2 filename.bz2
Convert text files to PostScript	enscript	enscript -o file.ps file

(*Continued*)

Format files for printing on a PostScript printer	a2ps	a2ps -o code.ps code.f
Printing and pagination filter for text files	pr	pr program.f90 > program.f90.pr
For transferring files between computers, use	scp	scp yourname@host:file file
‘scp’ (secure copy) or ‘sftp’ (secure ftp)		
		scp yourname@host:file .
		scp file yourname@host:.
		scp file yourname@host:file
	sftp	username@host
		pwd, cd subdir, ls, !ls, put, get, quit
To log on remotely, the preferred protocol is ‘ssh’	ssh	yourname@host

with a ‘-’ symbol in front. Individual arguments are separated by white space, that is, one or more spaces or tabs:

```
> cp file1 file2                                   ↩
> grep 'a string' file                             ↩
```

There are two arguments in both of the above examples. Note the use of single forward quotes needed when supplying the `grep` command with an argument (i.e. ‘`a string`’) which contains spaces. The command

```
> grep a string file
```

without quotes has three different arguments rather than just two, and thus has a completely different meaning.

1.3.1 Executables and paths

In Unix/Linux, a command such as `ls` or `cp` is usually a file, which is known to the system to be executable. To invoke the command, you must either type the absolute pathname of the executable file or ensure that the file can be found in one of the directories specified by your path. For the C-shell and bash shell, the current list of directories which constitute your path is maintained in the shell variable, `PATH`. To display the contents of this variable, type

```
> echo $PATH                                       ↩
```

The '$' mechanism is the standard way of evaluating shell variables and environment variables alike. The resulting output generated by the C-shell may look something like this,

```
/home/student/bin:/home/student/local/bin:/usr/local/
sbin
```

The order in which path components (that is, first `/home/student/bin`, then `/home/student/local/bin`, then `/usr/local/sbin`) appear in the path is important. When you invoke a command without using an absolute pathname, as for example

```
> ls
```

the system looks in each directory in your path, in the specified order, until it finds a file with the appropriate name. If no such file is found, the shell returns an error message. As an example, say you want to list all files and directories in a given directory. This is accomplished by typing `ls` at the shell prompt and hitting the return button. Instead of `ls`, however, say you erroneously type `list`, which does not exist on your machine. The shell therefore will return an error message such as

```
– bash: list: command not found
```

The path variable is typically set in your `~/.login` file and/or preferably your `~/.cshrc` or `~/.bashrc` files, which reside in your home directory. Examining `~/.cshrc` and `~/.bashrc` you should see lines like

```
export PATH=/usr/local/bin:/home/student/bin:$PATH
set path=($path /usr/local/bin $HOME/bin)
```

for the bash shell and the C-shell, respectively. These lines add the directories `/usr/local/bin` and `$HOME/bin` to the previous (system default) value of `PATH`. Also note the use of parentheses to assign a value containing whitespace to the shell variable. `HOME` is an environment variable which stores the name of the home directory. Thus

```
set path=($path /usr/local/bin ~/bin)
```

will have the same effect as

```
set path=($path /usr/local/bin $HOME/bin)
```

Control characters: The control characters CTRL-D, CTRL-C, and CTRL-Z have special meanings or uses within a shell. Below we shall familiarize ourselves with the actions and typical usages of these control characters. We shall use a caret ('^') to denote the CTRL key. Then, for instance,

```
> ^D
```

means pressing the (upper- or lower-case) D-key while holding down the CTRL (control) key. If you try the above example, you will notice that the shell does not 'echo' the ^D. This is typical of control characters. When you type ^D, the operating system sends all of the current lines that you have typed (but not the ^D itself) to the program (e.g. mail program, LaTeX) doing the read, which may echo the characters end-of-transmission (EOT). Other commands such as `cat`, for instance, will not echo anything. In almost all cases, however, you should be presented with the shell prompt. By default, the C-shell and bash shell exit when they encounter an end-of-file (EOF). So if you type ^D at a the shell prompt, the terminal will close automatically. This behavior can be changed by adding `set ignoreeof` to `~/.cshrc` for the C-shell and `export ignoreeof=1` to `~/.bashrc` for the bash shell.

The ^C interrupt kills (stops in a non-restartable fashion) commands (processes) which have been started from the command-line of a terminal window. This is particularly useful for commands which are taking much longer to execute or producing much more output to the terminal than anticipated. Many commands catch interrupts and you may sometimes have to type more than one to stop the command.

The ^Z interrupt suspends, i.e. stops in a restartable fashion, commands which have been started from the shell. This is useful as it is often convenient to temporarily halt execution of a command.

1.3.2 Special files

The following files, all of which reside in your home directory, have special purposes and you should familiarize yourself with their content. The first one is `.cshrc`. Commands in this file are executed each time a new C-shell is started. The second file to note is `.login`. Commands in this file are executed after those in `.cshrc` and only for login shells. When interacting with Unix/Linux via a window system, it is easy to start an interactive shell which is not a login shell, but for which you presumably want the same initialization procedures. Consequently, your `.login` should be kept as brief as possible and all your start-up commands should be put in `.cshrc` instead. Users using the bash shell rather than the C-shell should put all their the start-up commands in `.bashrc`.

Note that files whose name begins with a period ('.') are called hidden files. They are not shown in a standard listing generated with `ls`, but can be printed by adding the `-a` operand to the listing command, as shown here:

```
> ls —a                                                    ↩
```

Listing the names of all files in your home directory is accomplished with

```
> cd ; ls -a                                                    ↩
```

where we have introduced another piece of shell syntax, namely the ability to type multiple commands separated by semicolons (‘;’) on a single line. If one wants to list only the hidden files and hidden directories in a given directory, the following command is to be executed:

```
> ls -d .*                                                      ↩
```

where the `-d` operand guarantees that directories are listed as plain files (not searched recursively) and the asterisk (‘*’) stands for any number of characters.

Shell aliases: The syntax of many Unix/Linux commands is quite complicated and furthermore, the bare-bones version of some commands is less than ideal for interactive use, particularly by novices. The C-shell and bash shell provide a mechanism called aliasing which allows one to easily remedy these deficiencies in many cases. The basic syntax for aliasing is

```
 alias name definition
```

where `name` is the name (use the same considerations for choosing alias names as for filenames, i.e. avoid using special characters) of the alias and `definition` tells the shell what to do when you type `name` at the shell prompt, as if it was a command. The following examples give a basic idea how this works. More details can be found in the system’s manual pages by typing `man csh` for the C-shell, and `man bash` for the bash shell. A convenient re-definition of the standard listing command, for instance, is

```
% alias ls 'ls -FC'                            (for the C-shell)
> alias ls='ls -FC'                         (for the bash shell)
```

These aliases for the `ls` command uses the `-F` and `-C` options, which are described in the discussion of the `ls` command below. Note that single quotes in alias definitions are essential if the definitions contains white spaces. The commands

```
% alias rm 'rm -i'                                              ↩
% alias cp 'cp -i'                                              ↩
% alias mv 'mv -i'                                              ↩
```

define C-shell aliases for `rm`, `cp`, and `mv` which will request confirmation before attempting to remove, copy, or move each file, regardless of the file’s permissions. For the bash shell, the above shell commands read

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

Making use of aliases is highly recommended for novices and experts alike. To see a list of all current aliases for a given shell, simply type

```
> alias                                               ↩
```

Note that aliases defined interactively in a given shell exist only as long as the terminal session is open. To create aliases permanently, they need to be defined in `~/.aliases` or `~/.bashrc`, which are located in your home directory, or in `profile.local` which resides in the `/etc/` directory. The aliases are made available to shells with the `source` command, by typing

```
> source ~/.aliases                                   ↩
> source /etc/profile.local                           ↩
```

at the shell prompt. The source command tells the shell to execute the commands in the files supplied as arguments.

1.4 Basic commands

The following list is by no means exhaustive, but rather represents what we consider an essential base set of Unix/Linux commands with which you should familiarize yourself as soon as possible. Refer to the manual pages (see below) for additional information about these commands.

1.4.1 Getting help and information

Use `man`, which is short for manual, to display information about a specific Unix/Linux command. The `-k` option may be used in combination with `man` to display a list of commands which have something to do with a specific topic or keyword. For example, typing

```
> man -k xterm                                        ↩
```

returns all information found on the system about the X-terminal window. It cannot be overemphasized how important it is for users to become familiar with this command. Although the level of intelligibility for commands (especially for novices) varies widely, most basic commands are thoroughly described in the man pages, with usage examples in many cases. It helps to develop an ability to scan quickly through text looking for specific information you might feel to be of use. Typical usage examples include:

```
> man man
```

to obtain detailed information on the `man` command itself,

```
> man cp
```

for information on `cp`, and

```
> man -k 'working directory'
```

to obtain a list of commands having something to do with the topic `working directory`. The command `apropos`, found on most Unix/Linux systems, is essentially an alias for `man -k`.

1.4.2 Communicating with other computers

The OpenSSH secure shell client `ssh`, a remote login program, can be used to securely login to another computers on the Internet and perform command-line operations on them interactively. These computers could be physically located anywhere in the world. `ssh` is the most common way to access remote Linux and Unix-like machines. The typical usage of `ssh` is either

```
> ssh remote.host.name -l login_name                            ↩
```

or, alternatively,

```
> ssh login_name@remote.host.name                               ↩
```

which initiates the login of a user named `login_name` on the remote machine with the network ID `remote.host.name`. The `-l` option specifies the login name of the user on the remote machine. Let us look at an example. As `login_name` we pick `student`. The login session is then initiated by typing `ssh student@remote.host.name` at the shell prompt (>) of the local machine, as shown below:

```
> ssh student@remote.host.name (on the local machine)
student@remote.host.name's password: xxxxxx                      ↩
Login successfull from remote.host.name
student@remote >
```

If user student is known on the remote machine, he/she will be asked for the password. Hereupon the remote machine returns the shell prompt, which allows student to run programs on the remote machine. If the login attempt fails because of a wrong password or an incorrect username, a permission denied message will be printed and the failed login attempt will most likely be recorded on the remote machine. In the above example, commands processed at the local machine are shown in red typewriter font, while those processed at the remote machine are shown in black typewriter font. To leave the remote terminal window session, type exit at the shell prompt.

The Secure File Transfer Protocol (SFTP) enables secure file transfer capabilities between networked machines. It also provides remote file system management functionality, allowing users to list the contents of remote directories and to delete remote files. Below is an example which illustrates how SFTP is used to copy a file named thesis.pdf from the remote host remote.host.name to the local host local.host.name. The user name is again student, who has an account on the remote host:

```
> sftp student@remote.host.name (on the local machine)

student@remote.host.name's password:  xxxxxx                ↩
sftp> pwd                                                    ↩
Remote working directory:  /home/student
sftp> ls                                                     ↩
public  temporary  numerical_codes  Thesis
sftp> cd Thesis                                              ↩
sftp> pwd                                                    ↩
Remote working directory:  /home/student/Thesis
sftp> ls                                                     ↩
thesis.pdf
sftp> get thesis.pdf                                         ↩
Fetching /home/student/Thesis/thesis.pdf to thesis.pdf
/home/student/Thesis/thesis.pdf 100% 910KB 500.6KB/s
sftp> !ls                                                    ↩
thesis.pdf
sftp> quit                                                   ↩
```

As in the previous example, the command shown in red is typed on the local machine, and the commands and messages on the remote machine are in black. The commands ls, cd, and pwd are used to, respectively, list the files and directories, change directories, and print the name of the current directory on the remote machine. The transfer of a file, `thesis.pdf` in the current example, from the remote machine to the local machine is accomplished with the `get` command. Conversely, the command `put` is to be used if a file is sent from the local machine to the remote machine. The command `!ls` is used to list the files and directories on the local machine, without leaving the SFTP session. Similarly `lcd` and `lpwd` can be used to change the working directory and to display the current working directory on the local server. Submitting the `quit` command terminates the SFTP session, as shown in the example above.

The `sftp` program offers fairly extensive on-line help, which can be retrieved by typing

```
sftp> help                                                          ↩
```

or by submitting one of the following commands:

```
sftp> help bin                                                      ↩
sftp> help cd                                                       ↩
sftp> help lcd                                                      ↩
sftp> help put                                                      ↩
sftp> help get                                                      ↩
sftp> help prompt                                                   ↩
sftp> help mget                                                     ↩
```

1.4.3 Creating, manipulating, and viewing files and directories

The text editors which will be considered in this book are ‘vi’ and ‘Emacs’. The vi editor (short for visual editor) is a simple screen editor which is available on almost all Unix systems. ‘Emacs’ belongs to a family of text editors that are characterized by their enormous extensibility. Either of these two editors is perfectly suited to create, modify, and view text files at the level required for this course. Both editors are very popular among programmers, scientists, engineers, students, as well as system administrators. A brief introduction to vi and emacs is provided in chapter 2. Most often `vi`, or its improved version named `vim`, is started to edit a single file with the commands

```
> vi filename                                                       ↩
> vim filename                                                      ↩
```

Similarly, the command to start an Emacs session at the shell prompt is given by

```
> emacs filename                                                    ↩
```

The command `more` is used to view the contents of one or more files one page at a time. For example, executing the `more` command as

```
> more ~/.bashrc                                              ↩
## Source global definitions
if [-f /etc/bashrc ]; then
. /etc/bashrc
fi
## Source local definitions
if [-f /etc/profile.local ]; then
. /etc/profile.local
fi
alias rm='rm -i'
alias mv='mv -i'
alias cp='cp -i'
alias dir='ls -aF'
```

displays the first page of lines of the `.bashrc` configuration file, which is located in the home directory. The next page of lines (if any) is displayed by hitting the spacebar. Scrolling backward by one page in is accomplished by typing `b`. Forward scrolling by one page is done by typing d| and the command `q` quits viewing a file. Consult the man pages (`man more`) for the many other features of the `more` command.

The commands `lp` or `lpr` are used to print files. By default, files are sent to the system default printer, or to the printer specified in your PRINTER environment variable. The typical usage is

```
> lp -d laser print.ps                                        ↩
```

which prints postcript file `print.ps` at the printer named `laser`. If you want to print a regular text file or the source code of a numerical program such as Fortran or C++, it is highly recommended to convert these files first to postscript files using the `enscript` command. The typical usage is

```
enscript -o print.ps file.txt                                 ↩
enscript -o print.ps file.f90                                 ↩
enscript -o print.ps file.cpp                                 ↩
```

For detailed information about this command, type `man enscript`.

The commands cd and pwd are used to, respectively, change and display the current working directory. Below we show a summary of the commands that are typically used. Note the usage of semicolons to separate distinct Unix/Linux commands issued on the same line:

```
> cd                                                      ↩
> pwd                                                     ↩
/home/student
> cd ~; pwd                                               ↩
/home/fweber
> cd /tmp; pwd                                            ↩
/tmp
> cd ..; pwd                                              ↩
/
```

Recall that '..' refers to the parent directory of the working directory so that

```
> cd ..                                                   ↩
```

takes you up one level in the file system hierarchy.

The listing command ls is used to list the contents of one or more directories, as shown for the home directory in this example:

```
> cd                                                      ↩
> ls                                                      ↩
Desktop Downloads thesis numerical homework paper.pdf
```

The listing can be made more explicit by redefining the ls command as

```
> alias ls='ls -F'                                        ↩
```

which causes ls to append special characters, notably '*' for executables, '@' for links, and '/' for directories, to the names of certain files and directories. Then

```
> ls                                                      ↩
Desktop/ Downloads/ thesis/ numerical/ homework/ paper.pdf
```

which immediately reveals that `Desktop`, `Downloads`, `thesis`, `numerical`, and `homework` are directories and `paper.pdf` is a regular file. To display hidden files in directories, the `-a` option is to be used:

```
> cd ~; ls -aF                                                    ↩
.bashrc .bash_profile .local/ .profile .vim .xemacs
Desktop/ Downloads/ thesis/ numerical/ homework/
paper.pdf
```

Finally, using `ls` in combination with the `-l` option allows one to display file and directory information in long format:

```
> cd /numerical; ls -lF                                           ↩
-rwxr-x--- 15 student users 409 Aug 21 19:18 data
lrwxrwxrwx 11 student users 817 Mar 22 14:19 f77@ -> bu/
drwxr-xr-x 51 student users 170 Apr 20 12:34 f90/
drwxr-xr-x 17 student users 130 Aug 20 13:03 f2008/
drwxr-xr-x 70 student users 990 Feb 22 23:51 cpp/
```

The output in this case is worthy of a bit of explanation. First, observe that `ls` produces one line of output per file and directory listed. The first field in each listing consists of ten characters (i.e. letters and dashes) which are further subdivided as follows:

- The first character is either a '`-`' if the listing refers to a regular file, a '`d`' for a directory, and a '`l`' for a link.
- The next nine characters refer to 3 groups (user, group, other or world) of 3 characters each specifying read (`r`), write (`w`), and execute (`x`) permissions for the user (owner of the file), users in the owner's group, and all other users. A '`-`' in the permission field indicates that the particular permission is denied.

Thus, in the above example, `data` is a regular file, with read, write, and execute permissions enabled for the owner (user `student`), read and execute permissions enabled for the members belonging to group `users`, and read, write and execute denied for all other users. Note that you must have execute as well as read permissions for a directory in order to be able to change (`cd`) to this directory. See `chmod` below for more information on setting file permissions. Continuing to decipher the file listing, the next column in the above example lists the number of links to this file, then comes the name of the user who owns the file and the owner's group. This is followed by the size of the file in bytes, the date and time the file was last modified, and finally the name of the file. If any of the arguments to `ls` is a directory, then the contents of the directory is listed. Finally, we note that the `-R` option is used to recursively list sub-directories encountered in a given directory

```
> cd ~; pwd; ls                                          ↩
/home/student
Desktop/ Downloads/ thesis/ numerical/ homework/
paper.pdf
> ls -R /homework                                        ↩
instructions.txt
assignment.tex

homework//HW1:
code.f90
code.f90.ps
figures.ps
```

The command `mkdir` is used to make (create) directories. The following example illustrates how to create a directory named `tempdir` in a user's home directory:

```
> cd ~                                                   ↩
> mkdir tempdir                                          ↩
> cd tempdir; pwd                                        ↩
home/student/tempdir
```

If one wants to create a deep directory, i.e., a directory for which one or more parent directories do not exist, the `-p` option is to be used in combination with `mkdir`, which automatically creates parent directories when needed:

```
> cd ~                                                   ↩
> mkdir -p dir1/dir2/dir3/dir4                           ↩
> cd dir1/dir2/dir3/dir4; pwd                            ↩
home/student/dir1/dir2/dir3/dir4
```

In this case, the `mkdir` command creates the `~/dir1` directory first, followed successively by `~/dir1`, `~/dir1/dir2`, `~/dir1/dir2/dir3`, and finally `~/dir1/dir2/dir3/dir4`, all residing in the home directory, `~`, of user `student`.

Copying files is accomplished with the `cp` command. This command can be used to create an identical copy of a file, copy one or more files to different directories, or duplicate an entire directory structure. The simplest usage is

```
> cp file1 file2                                         ↩
```

which copies the contents of `file1` to `file2` in the current working directory. Assuming that `cp` is aliased to `cp -i`, which is highly recommended, the aliased command returns a prompt to the terminal window before a file will be copied that would overwrite an existing file. If the user's response from the terminal is 'y' the file copy is carried out. Typing 'n' cancels the file copy. Below is an example of how this works. Assuming that file2 already exists in the current working directory, the shell dialog may be as follows:

```
> cp −i file1 file2                                          ↩
overwrite file2?  (y/n [n])                                  ↩
not overwritten
```

For many systems, [n] is the default option, as shown above, in which case only a simple shell return (↩) is required to not overwrite file2. To copy one or more files to a different directory, the typical command usage is

```
> cp −i file1 file2 temporary/.                              ↩
```

which attempts to copy file1 and file2 to sub-directory temporary. If files with identical names already exist in temporary, prompts at the terminal window will be returned which allow the user to either overwrite or cancel the file copy. An example which overwrites existing files is shown here:

```
> cp −i file1 file2 temporary/.                              ↩
overwrite temporary/./file1?   (y/n [n]) y                   ↩
file1 → temporary/./file1
overwrite temporary/./file2?   (y/n [n]) y                   ↩
file2 → temporary/./file2
```

Finally, duplicating an entire directory structure is done by adding the '-r' (recursive) option to cp. This copies the entire directory hierarchy to a new directory, such as

```
> cp −ir temporary/. copy_temporary                          ↩
```

The command mv is used to rename files or to move files from one directory to another. Again, let us assume that mv is aliased to mv -i so that the user will be prompted if an existing file would be clobbered by the move command. Here is an example illustrating the usage of the mv command:

```
> ls                                                    ↩
fileA
> mv fileA fileB                                        ↩
> ls                                                    ↩
fileB
```

The following sequence of commands illustrates how files file1, file2, and file2 located in home/student/subdir1/subdir2/subdir3 can be moved up one level in the directory structure:

```
> pwd                                                   ↩
/home/student/subdir1/subdir2
> ls                                                    ↩
subdir3
> cd subdir3                                            ↩
> ls                                                    ↩
file1 file2 file3 file4
> mv file1 file2 file3 ../.                             ↩
> ls                                                    ↩
file4
> cd ..                                                 ↩
> pwd                                                   ↩
/home/student/subdir1/subdir2
> ls                                                    ↩
file1 file2 file3 subdir3
```

The rm command is used to remove (delete) files or directory hierarchies. The use of the alias rm -i, which requests confirmation (y for yes, n for no) before attempting to remove files or directories, is highly recommended. Note that once a file or directory has been removed in Unix/Linux there is essentially nothing you can do to restore them other than restoring a copy from a backup. In this example

```
> rm −i oldStuff.dat                                    ↩
remove oldStuff.dat?  y                                 ↩
oldStuff.dat
```

the remove command is used to delete a data file named oldStuff.dat once the action is confirmed with yes. The command

```
> rm file1 file2 file3                                  ↩
```

removes several files at once, and

```
> rm −r thisDir                                              ↩
```

removes the entire content of directory `thisDir`, including the directory itself. Be particularly careful when using the `-r` option as all files and directories will be irrevocably lost. Using the remove command without the `-r` option, that is, `> rm thisDir`, can not be used to remove `thisDir`. If submitted at the shell prompt, Unix/Linux will complain that `thisDir` is a directory and no action will be taken.

The command `chmod` is used to modify the permissions (file mode bits) of file. See the discussion of `ls` above for a brief introduction to file permissions and check the man pages for `ls` and `chmod` for additional information. Basically, file permissions control who can do what with files. This includes yourself (the user, `u`), users in your group (`g`), and the rest of the world (the others, `o`). The file mode bits include the read bit (`r`), write bit (`w`), and the execute bit (`x`). When a user creates a new file, the system sets the permissions (mode bits) of a file to default values which can be modified with the `umask` command (see `man umask` for more information). The default `umask` on many Unix/Linux systems is `022`, which means that newly created files are readable by everyone (i.e., the world), but only writable by the owner, as shown below:

```
> touch newFile                                              ↩
> ls −dl newFile                                             ↩
>−rw−r−−r−− 1 student users 0 Aug 25 12:55 newFile
```

To change the `umask` setting of the current shell to something else, say `077`, run

```
> umask 077                                                  ↩
```

which changes the file mode bits for any newly created file in that shell to `-rw-------`. On Unix/Linux machines, the defaults should be such that you can do anything you want to a file you have created, while the rest of the world (including fellow group members) normally has only read and, where appropriate, execute permission. As the man page will tell you, you can either specify permissions in numeric (octal) form or symbolically. The latter are more intuitive and easier to remember. Several useful examples are shown below. Let us begin with

```
> chmod go−rwx file.f90                                      ↩
```

which removes all permissions from `group` and `others`. A file listing therefore produces on the following terminal window output,

```
> ls −dl file.f90                                            ↩
> −rw−−−− 1 student users 33 Aug 13:09 file.f90
```

To make a file executable by everyone, the a option, which stands for all (i.e. user, group, and other) can be used,

```
> chmod a+x file.o                                                    ↩
```

To remove this permission from everyone, the a-x option would be used. Finally, as a last example, the command

```
> chmod u-w thesis.tex                                                ↩
```

removes the user's write permission to a file to prevent accidental modification of particularly valuable information, such as a thesis. As indicated above, file permissions are granted by putting a '+' sign after 'ugo' or a, and removed by putting a '-' sign there.

1.5 More on the C-shell

1.5.1 Shell variables

The C-shell (csh) maintains a list of local variables, some of which, such as path, term, and shell are always defined and serve specific purposes within the shell. Other variables, such as filec and ignoreeof are optionally defined and frequently control details of shell operation. Finally, you are free to define your own shell variables as you see fit, but beware of redefining existing variables. By convention, shell variables have all-lowercase names. To see a list of all currently defined C-shell variables, simply type

```
% set                                                                 ↩
```

or

```
% set | more                                                          ↩
```

at the C-shell prompt (%). Using more will display as many lines as fit on the screen and prompts the shell to wait for user input (i.e. ↩) to advance. To print the value of a particular variable, use the Unix/Linux echo command plus the fact that a $ symbol in front of a variable name causes the evaluation of that variable,

```
% echo $PATH                                                          ↩
```

To set the value of a shell variable use one of the following two ways,

```
% set thisvar=thisvalue                                          ↩
% echo $thisvar                                                  ↩
thisvalue
```

or

```
% set thisvarlist=(value1 value2 value3)                         ↩
% echo $thisvarlist                                              ↩
value1 value2 value3
```

Shell variables may be defined without being associated a specific value, as shown here:

```
% set somevar                                                    ↩
% echo $somevar                                                  ↩
```

The shell frequently uses this 'defined' mechanism to control enabling of certain features. To undefine a shell variable use `unset` as in

```
% unset somevar                                                  ↩
% echo $somevar                                                  ↩
somevar: Undefined variable.
```

The following is a list of some of the main shell variables (predefined and optional) and their functions:

- `path`: Stores the current path for resolving commands.
- `prompt`: The current shell prompt—what the shell displays when it is expecting input.
- `cwd`: Contains the name of the (current) working directory.
- `term`: Defines the terminal window type. If your terminal is acting strangely, the command

```
% set term=vt100; resize                                         ↩
COLUMNS=87;
LINES=23;
export COLUMNS LINES;
```

  often provides a quick fix.
- `noclobber`: When set, prevents existing files from being overwritten via output re-direction (see below).
- `filec`: When set, this enables file auto completion. Partially typing a filename, using an initial sequence which is unique among files in the working directory,

followed by hitting the TAB button will result in the system doing the rest of the typing of the filename for you.
- shell: Defines which particular shell you are using.
- ignoreeof: When set, this will disable shell-logout when ^D is typed.

1.5.2 Environment variables

Aside from shell variables discussed in section 1.5.1, Unix/Linux uses another type of variable, called an environment variable, which is often used for communication between the shell (not necessarily the C-shell) and other processes. By convention, environment variables have all-uppercase names. In the C-shell, you can display the value of all currently defined environment variables by typing

```
% env | more                                              ↩
```

Some environment variables, such as PATH are automatically derived from shell variables. Others have their values set, typically in ~/.cshrc or ~/.login, using the syntax

```
% setenv VARNAME value                                    ↩
```

Note that, unlike the case of shell variables and set, there is no '=' sign in the assignment. The values of individual environment variables may be displayed using the commands printenv or echo:

```
% printenv HOME                                           ↩
/home/student
% echo $HOME                                              ↩
/home/student
```

It should be noted that, as with shell variables, the '$' sign causes the evaluation of an environment variable. It is particularly notable that the values of environment variables defined in one shell are inherited by commands (including C and Fortran programs, and other shells) which are initiated from that shell. For this reason, environment variables are widely used to communicate information to Unix/Linux commands (applications). The DISPLAY environment variable is a canonical example of an environment variable. It tells X-applications which display (screen) to use for output. It is typically set on remote machines so that output appears on the local screen. For example, assuming you are remotely logged into host darwin from the console of your local machine magic, then, at the darwin prompt, you may want to type

```
% setenv DISPLAY darwin:0.0                                    ↩
```

after which all X-applications started on darwin will be displayed graphically on the local magic machine. If you encounter problems transporting windows from a remote machine to your local console, try typing xhost + at a shell prompt on the local machine. See man xhost for more information.

The HOME variable asks the shell to substitute the environment variable HOME. For example,

```
% cd $HOME/homework                                            ↩
```

allows you to change from any (sub) directory directly to homework, provided it exists in your home directory. Since HOME stand for your home directory (/), this command is equivalent to

```
% cd ~/homework                                                ↩
```

The PRINTER variable defines the default printer for use with lpr, lp, or programs such as enscript, which feed postscript files to a printer via lpr or lp. The default printers may be designated in ~/.cshrc by adding setenv PRINTER printername, or in ~/.bashrc by adding PRINTER=printername; export PRINTER, which sets the PRINTER variable for the C-shell and bash shell, respectively.

1.5.3 C-shell pattern matching

The C-shell provides facilities which allow you to concisely refer to one or more files whose names match a given pattern. The process of translating patterns to actual filenames or pathnames is known as filename/pathname expansion, or globbing. The name expansion expands the ‘*’, ‘?’, and a pattern list ‘[...]’ when you type them as part of a command. For example,

```
% *.ps                                                         ↩
```

lists all postscript files in a given directory, where the ‘*’ acts as a placeholder for any string of characters. Replacing the asterisk with a question mark in the above command, that is,

```
% ?.ps                                                         ↩
```

causes the shell to list all postscript files with only one-character filenames, such as a.ps or 2.ps. Pattern lists [...] are constructed using plain text strings sandwiched between square brackets, such as

```
% ls [A–Z]*.ps                                                    ↩
```

which lists all postscript files that start with any capital letter. If desired, these files could then be moved from the current working directory to ~/plots by typing

```
% mv [A–Z]*.ps ~/plots/                                           ↩
```

The command

```
% rm [A–Z]*.ps                                                    ↩
```

can be used to remove all files whose names begin with a capital letter. Submitting

```
% mv *.f90 ../f90Codes/                                           ↩
```

at the Unix/Linux shell prompt moves all files with extension .f90 to directory f90Codes, where the double period refers to the parent directory of f90Codes, i.e. the directory that contains f90Codes.

These are not the only forms of wildcards supported by csh or bash. Another useful wildcard, for instance, is the pattern list [a-z].ps which selects all postscript files whose names begin with a lower-case letter. The pattern list [^a-z], which filters out any single character not contained in the specified range, could be used to list only those files and directories that begin with a capital letter or a number, and the command

```
% [^b–z,A–Z]*                                                     ↩
```

will list all files and directories whose names begin with an ‘a’. Everything else would not be shown. The command

```
% ls ?????                                                        ↩
a.pdf
```

lists all regular (not hidden) files and directories whose names contain precisely five characters, such as for a.pdf. Last but not least, we mention that the command

```
% mv *.f *.for
```

will not rename all files ending with .f to files with the same prefixes, but ending in .for, as is the case for some other operating systems. This is easily understood by noting that file expansion occurs before the final argument list is passed along to the mv command. If there are no .for files in the working directory, *.for will expand to nothing and the shell command will be identical to

```
% mv *.f,
```

which is something very different from what was intended.

1.5.4 Using the C-shell history and event mechanisms

The C-shell maintains a numbered history of previously entered command lines. Because each line may consist of more than one distinct command (separated by a semicolon), the lines are called events rather than simply commands. To view the shell history, type

```
% history                                                    ↩
```

after entering a few commands at the shell prompt. Although `bash`, which I assume you are using, allows you to work back through the command history using the up-arrow and down-arrow keys, the following event designators for recalling and modifying events are still useful, in particular if the event number is part of the shell prompt, as is the case for the initial set-up on many Linux machines. The command

```
% !!                                                         ↩
```

causes the shell to repeat the previous command line, while

```
% !22                                                        ↩
```

will repeat the command with line number 22. Unix/Linux users often refer to an exclamation point ('`!`') as 'bang'. To repeat the most recently issued command line which started with an 'a', type

```
% !a                                                         ↩
```

An initial sub-string of length greater than one can be used for more specificity. The command

```
% !?b                                                        ↩
```

is used to repeat the most recently issued command line which contains 'b'. Any string of characters can be used after the question mark.

1.5.5 Standard input, standard output, and standard error

Every program run from a shell automatically opens three files (data streams), which are standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`). These files provide the primary means of communications between the programs. They exist for as long as a given process runs from a shell. The standard input file provides a way to send data to a process. As a default, standard input is read from the terminal keyboard. The standard output provides a means for the program to output data. As a default, standard output is written to the terminal display screen. The standard error is where the program reports any errors encountered during execution. By default, the standard error is written to the terminal display, too. Below, we use the `cat` command with no arguments to illustrate how `stdin` and `stdout` work:

```
% cat                                                      ↩
something                                                  ↩
something
something else                                             ↩
something else
^D
```

Here, the command `cat` run from a shell reads the lines marked red from `stdin` (i.e., the terminal window) and writes them, shown in blue, to `stdout` (also the terminal window). In other words, every line that is typed by the user is echoed by the command. A command, such as `cat`, which reads from `stdin` and writes to `stdout` is known as a filter.

1.5.6 Redirecting input and output

The power and flexibility of the `stdin` and `stdout` mechanism becomes apparent when input and output is redirection, which is implemented in the C-shell and the bash shell. As the name suggests, redirection means that `stdin` and/or `stdout` are associated with targets other than the terminal display. Input redirection is accomplished using the '<' (less than) character which is followed by the name of a file from which the input is to be read or extracted. Thus, the command line

```
% cat < input.dat                                          ↩
```

causes the contents of the file `input.dat` to be used as input for the `cat` command. If the content of `input.dat` is given by

```
% more input.dat                                           ↩
1
2
3
4
```

then feeding these numbers to `cat` leads to the following terminal display:

1.5. More on the C-shell

```
% cat < input.dat                                    ↩
1
2
3
4
```

Output redirection is accomplished by using the '>' (greater than) character, again followed by the name of a file to which the (standard) output of the command is to be written. Thus

```
% cat > output.dat                                   ↩
```

will cause `cat` to read lines from the terminal window and copy them to the file `output.dat`. Care must be exercised when using output redirection since one of the first things which will happen in the above example is that the file `output.dat` will be clobbered. If the shell variable `noclobber` is set (strongly recommended for novices), then output will not be allowed to be redirected to an already existing file. Thus, in the above example, if `output.dat` already exists, the shell would respond as follows,

```
% cat > output.dat                                   ↩
output.dat: File exists
```

and the command would be aborted. The standard output from a command can also be appended to a file using the two-character sequence '>>' (no intervening spaces). Thus

```
% cat >> existing_file.dat                           ↩
```

will append lines typed at the terminal to the end of `existing_file.dat`. From time to time it is convenient to be able to throw away the standard output of a command. Unix/Linux systems have a special file called `/dev/null` which is ideally suited for this purpose. Output redirection to this file, as shown in this example,

```
% cat input.dat > /dev/null                          ↩
%
```

causes the `stdout` output to disappear entirely from the command line terminal. Only the shell prompt is returned on the terminal window.

1.5.7 Pipelines

It is then often possible to combine commands (programs) on the command line so that the standard output from one command is fed directly into the standard input of another. In this case we say that the output of the first command is piped into the input of the other. Here is an example:

```
% ls -l | wc                                                    ↩
63 588 3964
```

The `-l` option tells the listing command `ls` to show regular files and directories, one per line. The command `wc` (which stands for word count) when invoked with no arguments, reads `stdin` until an end-of-file (EOF) is encountered and then prints three numbers: (1) the total number of lines in the input, (2) the total number of words in the input, and (3) the total number of characters in the input. For the above example, these numbers are 63, 588 and 3964, respectively. The pipe symbol '|' tells the shell to connect the standard output of `ls` to the standard input of the `wc` command. The entire `ls -l` | `wc` construct is known as a pipeline. The first number (i.e. 63) which appears on the standard output is thus simply the number of regular files and directories in the current directory, where the listing is being created.

Pipelines can be made as long as desired, and once you know a few Unix/Linux commands and have mastered the basics of the C-shell history mechanism, you can easily accomplish some fairly sophisticated tasks by building up multi-stage pipelines.

A powerful Unix/Linux tool which searches for a matching regular expression against text in a file, multiple files, or a stream of input is the `grep` command. It searches for the pattern of text that is specified on the command line and prints output for the user. `grep`, which loosely stands for (g)lobal search for (r)egular (e) xpression with (p)rint, has the following general syntax,

```
grep [options] regexp [file1 file2 ...]
```

where `regexp`, which stands for regular expression, is a string that is used to describe several sequences of characters. Invoking `grep` with just a regular `regexp` as the only argument,

```
% grep regexp                                                   ↩
```

will read lines from `stdin`, usually the terminal window, and echo only those lines which contain the string `regexp`. If one or more file arguments are supplied along with `regexp`, then `grep` will search all those files for lines matching `regexp`, and print the matching lines to standard output, which is usually the terminal window again. Thus

```
% grep thesis *                                                 ↩
```

will print all the lines of all the regular files in the current working directory which contain the string thesis. Files in subdirectories residing in the working directory will not be searched, however. Recall that the '*' wildcard represents every string. So it can be used as the argument file for file causing the shell to search for thesis in all files in the current directory. A few more useful options to grep are worth mentioning. The first is -i, which tells grep to perform a case insensitive pattern matching. By default, grep is case sensitive. Thus

```
% grep -i thesis mynotes                                    ↩
```

will print all lines of the text file mynotes which contain 'thesis' or 'Thesis' or 'THes', etc. Second, the -v option instructs grep to print all lines which do not match the pattern. An example of this is shown here,

```
% grep -v thesis mynotes                                    ↩
```

which will print all lines of text of mynotes which do not contain any of the symbols contained in student. Finally, the -n option tells grep to include a line number at the beginning of each line that is being printed. Thus

```
% grep -in thes mynotes                                      ↩
133:  Notes regarding my thesis:
325:  The date of the thesis defense is still unclear.
910:  The thesis committee consists of four members.
```

searches the file mynotes for the case insensitive pattern thes and prints all lines, together with line numbers in the first column, which contain the strings 'thes', 'Thes', 'tHes', etc. Note that multiple options can be specified with a single '-' sign followed by a string of option letters with no intervening blanks.

Next we show a few, slightly more complicated examples of how grep can be used to find strings of text. Note that when supplying a regular expression that contains characters such as '*', '#', '?', '[', or '!', which are special to the shell, the regular expression should be surrounded by single quotes to prevent shell interpretation of the shell characters. In fact, a user will not go wrong by always enclosing the regular expression in single (or double) quotes, as shown in this example:

```
% grep -owE '^[[:alnum:]]{7}' mynotes                        ↩
```

This will search for, and print on the terminal window, all alphanumeric strings in mynotes that are exactly seven characters long. The command

```
% grep 's' mynotes | grep 't'                               ↩
```

prints all lines of mynotes which contain at least one 's' and one 't', such as lines containing 'student' or 'thus'. Note the use of the pipe symbol used to redirect the stdout from the first grep to the stdin of the second grep. The command

```
% grep -v '^#' mynotes > output                                    ↩
```

extracts all lines from file mynotes which do not have a '#' in the first column and writes them to a file named output. Pattern matching using regular expressions, as discussed just above, is a powerful tool. But it can be made even more powerful when combining it with certain extensions. Many of these extensions are implemented in a relative of grep, known as egrep. Details about egrep can be found in the man pages by typing man egrep.

1.5.8 Usage of quotes

Most shells, including the C-shell and the bash shell, use three different types of quotes found on every standard keyboard. These are regular quotes (') also known as forward quotes, single quotes, or just quote, double quotes ("), and backward quotes (') also referred to as just back quotes. They serve distinct roles on Unix/Linux machines, which will be discussed here.

Single quotes: We have already encountered several situations where forward quotes have been used to quote variables. In essence, they inhibit shell evaluation of special characters and/or constructs. Here is an example. In a terminal session, let us assign variable a a numerical value of 100 and then print the value of a with the echo $a command,

```
% set a=100                                                            ↩
% echo $a                                                              ↩
100
```

Next we assign the value of $a to the new variable b,

```
% set b=$a                                                             ↩
% echo $b                                                              ↩
100
```

and use echo $b to verify the value of $b. Now let us repeat the last steps but with $a put in quotes,

```
% set b='$a'                                                           ↩
% echo $b                                                              ↩
$a
```

which protects $a from shell evaluation. The command echo $b therefore does not return 100 but rather $a. Single quotes are commonly used to assign a shell variable a value which contains whitespace(s), or to protect command arguments which contain characters special to the shell (see the discussion of grep).

Double quotes: Double quotes function in much the same way as forward quotes, except that the shell looks inside them and evaluates both any references to the values of shell variables as well as anything sandwiched within back-quotes (see discussion of backward quotes below). An example is shown here:

```
% set a = 200                                    ↩
% echo $a                                        ↩
% 200
% set string="The value of a is $a"              ↩
% echo $string                                   ↩
The value of a is 200
% set string='The value of a is $a'              ↩
The value of a is $a
```

The first line assigns a numerical value to variable a, which is then printed on the terminal window. This is followed by the set string command line, which assigns a text message plus a numerical value, carried by a, to string. Thus echo $string returns the assigned text message but with 200 substituted for $a. As shown by the last two lines, this is not the case if single quotes are used, in which case the text message as sandwiched between the single quotes is returned to the terminal.

Backward quotes: The shell uses back-quotes to provide a powerful mechanism for capturing the standard output of a Unix/Linux command (or, more generally, a sequence of Unix/Linux commands) as a string which can then be assigned to a shell variable or used as an argument for another command. Specifically, when the shell encounters a string enclosed in back-quotes, it attempts to evaluate the string as a Unix/Linux command, precisely as if the string had been entered at the shell prompt, and returns the standard output of the command as a string. In effect, the output of the command is substituted for the string and the enclosing back-quotes. Here are a few simple examples:

```
% date                                           ↩
Sat Sep 2 13:16:22 PST 2017
% set current_date_and_time='date'               ↩
% echo $current_date_and_time                    ↩
Sat Sep 2 13:16:22 PST 2017
```

The date command returns the current date and time to the window terminal. The set command is used to assign the current date and time to the variable current_date_and_time, which is then echoed back to the terminal.

Full list of references

Chapter 3

[1] Etter D M 1995 *Fortran 90 For Engineers* 1st edn (New York: Wiley)
[2] Chapman S J 2007 *Fortran 95/2003 for Scientists and Engineers* 3rd edn (New York: McGraw-Hill)
[3] Chapman S J 2003 *Fortran 90/95 for Scientists and Engineers* 2nd edn (New York: McGraw-Hill)
[4] https://en.wikipedia.org/wiki/Fortran

Chapter 4

[1] Press W H, Flannery B P, Teukolsky S A and Vetterling W T 1992 *Numerical Recipes in FORTRAN 77: The Art of Scientific Computing* 2nd edn (Cambridge: Cambridge University Press)
[2] Press W H, Teukolsky S A, Vetterling W T and Flannery B P 2007 *Numerical Recipes: The Art of Scientific Computing* 3rd edn (Cambridge: Cambridge University Press)

Chapter 6

[1] Etter D M and Hayton J 1997 *Structured Fortran 77 for Engineers and Scientists* 5th edn (New York: Wiley)
[2] Etter D M 1995 *Fortran 90 For Engineers* 1st edn (New York: Wiley)
[3] Nahvi M and Edminister J 2017 *Schaum's Outlines of Electric Circuits* 7th edn (New York: McGraw-Hill)

Chapter 7

[1] Tolman R C 1939 *Phys. Rev.* **55** 364
[2] Oppenheimer J R and Volkoff G M 1939 *Phys. Rev.* **55** 374
[3] Chodos A, Jaffe, Johnson K, Thorne C B and Weisskopf V F 1974 *Phys. Rev.* D **9** 3471
[4] Chodos A, Jaffe R L, Johnson K and Thorne C B 1974 *Phys. Rev.* D **10** 2599
[5] Nahvi M and Edminister J 2017 *Schaum's Outlines of Electric Circuits* 7th edn (New York: McGraw-Hill)