IOPscience iopscience.iop.org

This content has been downloaded from IOPscience. Please scroll down to see the full text.

Download details:

IP Address: 18.117.73.223
This content was downloaded on 05/05/2024 at 17:15

Please note that terms and conditions apply.

You may also like:

Full Field Optical Metrology and Applications

Adsorption Applications for Environmental Sustainability

A Raman Spectroscopic Study of Radiation-Grafted Anion-Exchange Membranes Containing Different Anions
Rachida Bance-Soualhi, Carol Crean, Henryk Herman et al.

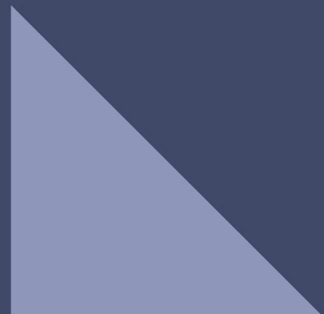
Special issue on applied neurodynamics: from neural dynamics to neural engineering
Hillel J Chiel and Peter J Thomas

Gas-Selective Semiconducting Oxide Nanowires from Novel Processing Methods
Anthony Annerino and Perena Gouma

IOP Concise Physics | A Morgan & Claypool Publication

Introduction to Computational Physics for Undergraduates

Omair Zubairi
Fridolin Weber



IOP ebooks

Introduction to Computational Physics for Undergraduates

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

Copyright © 2018 Morgan & Claypool Publishers

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the publisher, or as expressly permitted by law or under terms agreed with the appropriate rights organization. Multiple copying is permitted in accordance with the terms of licences issued by the Copyright Licensing Agency, the Copyright Clearance Centre and other reproduction rights organisations.

Rights & Permissions
To obtain permission to re-use copyrighted material from Morgan & Claypool Publishers, please contact info@morganclaypool.com.

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 ix

Acknowledgments x

Author biographies xi

1 The Linux/Unix operating system 1-1

1.1 Introduction 1-1
1.2 Files and directories 1-2
 1.2.1 Pathnames and working directories 1-2
 1.2.2 Filenames 1-5
1.3 Overview of Unix/Linux commands 1-6
 1.3.1 Executables and paths 1-8
 1.3.2 Special files 1-10
1.4 Basic commands 1-12
 1.4.1 Getting help and information 1-12
 1.4.2 Communicating with other computers 1-13
 1.4.3 Creating, manipulating, and viewing files and directories 1-15
1.5 More on the C-shell 1-23
 1.5.1 Shell variables 1-23
 1.5.2 Environment variables 1-25
 1.5.3 C-shell pattern matching 1-26
 1.5.4 Using the C-shell history and event mechanisms 1-28
 1.5.5 Standard input, standard output, and standard error 1-28
 1.5.6 Redirecting input and output 1-29
 1.5.7 Pipelines 1-31
 1.5.8 Usage of quotes 1-33

2 Text editors 2-1

2.1 Vi 2-1
2.2 Emacs 2-3

3 The Fortran 90 programming language 3-1

3.1 Compilers 3-1
 3.1.1 File extensions and compiling commands 3-2
3.2 Program layout 3-3

3.3 Variable declaration 3-5
3.3.1 Naming conventions 3-6
3.3.2 Data types 3-6
3.4 Basic expressions 3-8
3.4.1 Arithmetic operators and expressions 3-8
3.4.2 Relational operators 3-8
3.4.3 Logical expressions 3-9
3.5 Input and output 3-10
3.5.1 The READ statement 3-10
3.5.2 The WRITE statement 3-10
3.5.3 The FORMAT specification 3-11
3.5.4 File input and output (I/O) 3-12
3.6 Control structures 3-12
3.6.1 IF-blocks 3-12
3.6.2 DO loops 3-14
3.6.3 Nested loops 3-16
3.7 Modular programming 3-17
3.7.1 Intrinsic functions 3-17
3.7.2 Intrinsic subroutines 3-17
3.7.3 External functions 3-18
3.7.4 External subroutines 3-19
3.7.5 Program units 3-20
3.7.6 Internal procedures 3-21
3.7.7 External procedures 3-21
3.7.8 Modules 3-22
3.8 Arrays 3-23
3.8.1 Declaration of arrays 3-23
3.8.2 Vectors 3-23
3.8.3 Using arrays 3-24
3.8.4 Array operations 3-24
3.8.5 Elemental functions 3-25
3.8.6 The WHERE statement 3-25
3.8.7 FORALL (Fortran 95) 3-27
3.8.8 Array intrinsic functions 3-28
3.8.9 Allocatable arrays 3-28
3.8.10 Pointers 3-29
References 3-30

4 Numerical techniques 4-1

4.1 Curve fitting—method of least squares 4-1

4.1.1 The linear least-squares approximation 4-1

4.1.2 The quadratic least-squares approximation 4-3

4.2 Numerical differentiation 4-3

4.3 Numerical integration 4-4

4.3.1 The trapezoidal rule 4-5

4.3.2 Simpson's rule 4-6

4.4 Matrix operations 4-8

4.5 Finding roots 4-9

4.6 Solving ordinary differential equations 4-10

4.6.1 The Euler method 4-11

4.6.2 The midpoint method 4-12

4.6.3 The Runge–Kutta method 4-12

4.6.4 Boundary value problems 4-15

5 Problem solving methodologies 5-1

5.1 General guidelines 5-2

5.2 Projectile motion example 5-2

6 Worksheet assignments 6-1

6.1 Coding a mathematical expression 6-1

6.2 Comparing two functions 6-1

6.3 Bessel functions of the first kind 6-2

6.4 Logical IF statements 6-3

6.5 Lead concentration in humans (data analytics) 6-3

6.6 Nested DO loops and double summations 6-5

6.7 Ionic crystals 6-6

6.8 Least-squares fit 6-8

6.9 Numerical derivatives 6-9

6.10 Numerical integration 6-10

6.11 Finding roots of a nonlinear equation 6-10

6.12 Ordinary differential equations 6-11

6.13 Projectile in a viscous medium 6-11

6.14 Damped harmonic oscillator 6-13

6.15 RLC circuit 6-14

7 Homework assignments **7-1**
7.1 Fresnel coefficients 7-1
7.2 Earth atmosphere model 7-2
7.3 Magnetic permeability 7-3
7.4 Maxwell–Boltzmann distribution 7-4
7.5 Kinetic friction 7-5
7.6 Compton scattering 7-6
7.7 Radioactive decay 7-6
7.8 Halley's comet 7-7
7.9 Rocket equation 7-8
7.10 Hydrostatic equilibrium and relativistic stars 7-9
7.11 Massive stars 7-11
7.12 Isothermal gas spheres 7-11
7.13 Proton in constant electric and magnetic fields 7-12
7.14 Square voltage pulse applied to a RC circuit 7-13
7.15 Mutual inductance of two coils 7-14

Appendices

A Summary of Fortran features **A-1**

B Plotting using Python **B-1**

C Fortran 90 sample program illustrating good programming **C-1**

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.

Omair Zubairi and Fridolin Weber

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

 © Morgan & Claypool Publishers 2018

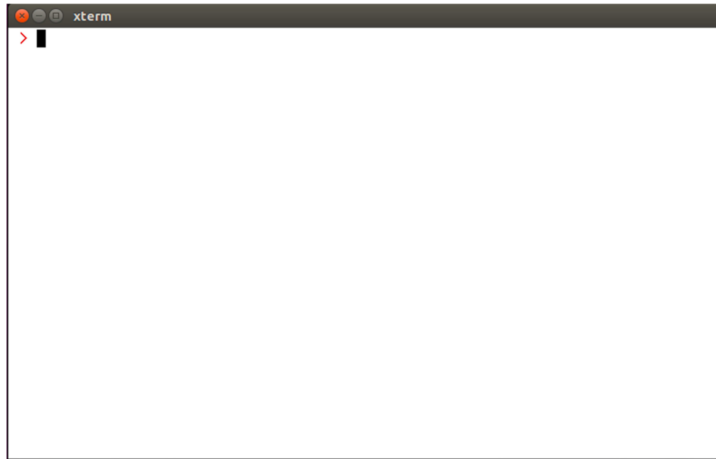


Figure 1.1. The X-terminal window on a machine running Ubuntu Linux.

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 postcript 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 -1 | 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.

Omair Zubairi and Fridolin Weber

Chapter 2

Text editors

All computer programs are written in some sort of text editor. There are many text editors across multiple platforms. Notepad++ is very popular on Windows, while 'vi' is the most widely used text editor on the Linux/Unix operating system. Due to the ease of the internet, text editors are widely available. Some others which can be used across multiple platforms include gedit, jedit, and atom, each with their unique niches. In this text we focus on two of the most versatile and widely used text editors, 'vi' or 'Vim' (vi Improved) and Emacs.

2.1 Vi

'vi' is a widely used text editor for UNIX systems. It is often available when other editors are not. vi does not make use of a user-interface menu nor does it have on-line help. You can learn about vi by using the man command. vi is a Unix command and therefore is case sensitive. Some vi commands are issued in UPPER-CASE. Be careful to make the distinction between upper-case commands and lower-case commands to issue the appropriate command. To create a new file or edit an existing one, you invoke the vi editor by keying: vi filename. The vi editor has two modes: *command mode* and *insert mode.* In command mode you can position the cursor or issue a vi command. The last line on the screen displays the name and size of your file.

Command mode allows you to position your cursor or issue a command. To issue a command in vi you use the ':' (colon) to precede the command. You enter the text mode by keying either an 'i' (insert) or an 'o' (open). In text mode you enter your text into a file. You use the <ESC> key (escape) to exit the text mode. There are several ways to save files and leave vi. To quit without saving the file key: ':q!'. To save the file and continuing editing key: ':w'. To save the file and quit vi key: ':wq'. After you have saved your file, you can also exit vi by keying: ':q'. The following list shows some of the most commonly used vi editing commands.

doi:10.1088/978-1-6817-4896-2ch2 © Morgan & Claypool Publishers 2018

Window movements	
<CTRL> d	scroll down
<CTRL> u	scroll up
<CTRL> b	page backward
<CTRL> f	page forward
1 G	go to first line
G	go to last line

Cursor movements	
H	home (upper left corner)
L	lower left corner
h	back a character
j	down a line
k	up a line
ˆ	beginning of line
$	end of line

Input	
a	append after cursor
i	insert before cursor
o	open line below
O	open line above

Deletion	
dd	delete current line
x	delete current character

Undo u	undo last change
U	undo all changes on-line

Rearrangement	
yy or Y	yank (copy) line to general buffer
yw	yank word to buffer
”ap	put text from buffer a after cursor
p	put general buffer after cursor
J	join lines

Search and replace	
/string/	line that contains 'string'
>%	entire file

2.2 Emacs

'Emacs' is an extensible, customizable text editor. Its features include content-sensitive editing modes, including syntax coloring, for a wide variety of file types including plain text, source code, and HTML. The following list shows some of the most commonly used Emacs editing commands (C = Control, M = Meta = Alt| Esc)

Basics	
C-x C-f	find file i.e. open/create a file in buffer
C-x C-s	save the file
C-x C-w	write the text to an alternative name
C-x C-v	find alternative file
C-x i	insert file at cursor position
C-x b	create/switch buffers
C-x C-b	show buffer list
C-x k	kill buffer
C-z	suspend emacs
C-X C-c	close down Emacs

Basic movement	
C-f	forward char
C-b	backward char
C-p	previous line
C-n	ext line
M-f	forward one word
M-b	backward one word
C-a	beginning of line
C-e	end of line
C-v	one page up
M-v	scroll down one page
M-<	beginning of text
M->	end of text

Editing	
M-n	repeat the following command n times
C-u	repeat the following command four times
C-u	*n* repeat *n* times
C-d	delete a char
M-d	delete word
M-Del	delete word backwards
C-k	kill line
C-Space	set beginning mark (for region marking for example)
C-W	kill (delete) the marked region region
M-W	copy the marked region
C-y	yank (paste) the copied/killed region/line
M-y	yank earlier text (cycle through kill buffer)
C-x C-x	exchange cursor and mark
C-t	transpose two chars
M-t	transpose two words
C-x C-t	transpose lines
M-u	make letters upper-case in word from cursor position to end
M-c	simply make first letter in word upper-case
M-l	opposite to M-u

Important	
C-g	quit the running/entered command
C-x	undo previous action
M-x	revert-buffer RETURN undo all changes since last save
M-x	recover-file RETURN recover text from an autosave-file
M-x	recover-session RETURN if you edited several files

Search/replace	
C-s	search forward
C-r	search backward
C-g	return to where search started (if you are still in search mode)
M-%	query replace
M-x	query–replace–regexp search and replace

Window commands	
C-x 2	split window vertically
C-x o	change to other window
C-x 0	delete window
C-x 1	close all windows except the one the cursor is in

Omair Zubairi and Fridolin Weber

Chapter 3

The Fortran 90 programming language

Fortran [1–3], derived from FORmula TRANSlation, is a powerful and widely used programming language, which was designed specifically for numerical applications. The language was invented by a team led by John Backus working for IBM in the early 1950s. Successive versions have added support for structured programming and processing of character-based data (FORTRAN 77), array programming, modular programming and generic programming (Fortran 90), high performance Fortran (Fortran 95), object-oriented programming (Fortran 2003) and concurrent programming (Fortran 2008) [4]. These features expand the Fortran language substantially and allow elaborate code to be written easily.

3.1 Compilers

Any source code written in the Fortran programming language needs to be compiled from another computer program written in some other language. A compiler is a computer program which transforms source code into a language which the computer can understand. This is usually known as 'compiling your source code'. Your source code when compiled will create another computer program usually referred to as an 'executable'. The program user will then 'run' the executable program which in turn will produce the desired results from the source code. Fortran has several excellent compilers such as

- ifort (Intel Fortran compiler)
- pgi (Portland Group Inc. compiler)
- NAG (Numerical algorithms Group compiler)
- af90 & af95 (Absoft Fortran Compiler)
- gfortran (GNU Fortran compiler)

doi:10.1088/978-1-6817-4896-2ch3

© Morgan & Claypool Publishers 2018

where each of these compilers has its own niche and is designed for various purposes. In this text, we will focus our attention on the 'gfortran' open source compiler which is readily available on multiple platforms and operating systems. The 'gfortran' compiler is also one of the more restrictive compilers allowing programmers to write code which will be compiler independent.

3.1.1 File extensions and compiling commands

The standard file extension for any Fortran 90 program is simply '.f90'. Using a text editor such as Vim, a Fortran program can be easily created in the Linux environment via the terminal command:

```
> vim filename.f90
```

Once, you have created and written your Fortran source code. The next step is to compile and run your program. To compile your Fortran program, the command requires a few things

- the type of compiler (i.e. gfortran);
- flags (for specifically naming your executable and for optimization purposes);
- executable name (i.e. filename.x); and
- source code (i.e. filename.f90).

The four mentioned items are all typed in one line of your Linux terminal such as

```
> gfortran -o filename.x filename.f90
```

The command above will create the executable `filename.x`, and thus to run this executable, simply type in the command:

```
> ./filename.x
```

The flag option '-o' in the example above is specifically for naming conventions of your executable. The file 'filename.x' is the executable you will be creating from your 'filename.f90' source code. It is important to note that the name of your executable can be anything and does not explicitly require a file extension; however, most programmers will use some sort of extension to distinguish the executable from the source code.

If your source code does not need any external libraries or you simply have only one Fortran file, you can also compile this one Fortran file via the command:

```
> gfortran filename.f90 ↩
```

which will result in an automatically created executable named a.out in the Linux/ Unix environment or a.exe on Windows platforms To run this executable, the same command applies as before

```
> ./a.out ↩
```

Every single time you compile, the automatic executables shown here will be overwritten and thus, one has to mindful of compilations of multiple programs.

The important thing to note is that the Linux compiler supports some features (not all) that are in the 90 standard, and which are very important to good programming. In particular, it supports the Fortran 90 structure of a Do–END DO loop, the type declaration statement with the double colon :: syntax, and the standard relational operators instead of the Fortran 77 verbose substitutions.

3.2 Program layout

Programs are usually written by using some text editor. Readability and cohesive flow throughout a program is not only important to the programmer, but is also vital for external collaborations with multiple parties. The guidelines listed below serve as good practice for writing computer programs.

- Start your code with some sort of prologue (describing your code).
- Give compiler directions (i.e. how to compile and run your code).
- Put Fortran key words and intrinsic functions in upper-case or at least have the starting letter capitalized.
- Put variable names in lower-case.
- Use descriptive variable names.
- Use blank lines to improve readability.
- Use multiple lines to improve readability (i.e. do not use one long line of code).
- Indent your code when possible.
- Last but not least, have a running commentary describing different parts of your code.

As you write more computer programs, you will start to have your own unique style of coding and you may not want to follow these rules explicitly; however, the general gist of good efficient programming lies within these general guidelines listed above.

As new text editors have emerged, Fortran 90 has evolved to meet these standards. A few important improvements include:

- A line may contain up to 132 characters and may contain more than one Fortran statement provided a semicolon separates each successive pair of statements.
- Blanks are significant.
- A trailing ampersand & indicates a statement is continued on the next line (a maximum of 39 continuation lines is allowed). Note that comments therefore cannot be continued, and a separate exclamation mark (!) must be used for each comment line.
- Statement labels consist of up to five consecutive digits preceding the command.
- Comment lines must begin with the exclamation mark !. Also, trailing comments (after a command) are allowed and also go after !.

The general structure of a Fortran 90 program looks something like this:

```
Declare Modules
PROGRAM name
!Comments and program information
    Variable Declarations and/or External Functions
    Body of program
END PROGRAM name
Declarations of user made functions
```

Two sample Fortran 90 programs are listed below for your reference. The first program '`PROGRAM hello`' is a standard program which outputs to terminal 'Hello World'. The second program '`PROGRAM math`' performs some basic mathematical operations and outputs the results to the terminal.

```
PROGRAM hello
!This simple program will output "Hello World."
    !print statement (output to terminal)
    Print*, "Hello World"
END PROGRAM hello
```

```
PROGRAM math
!Simple Fortran program to take in some variables and
!perform basic !mathematical operations.

!Explicitly define all variables:
Implicit None

!Variable declarations:
Real :: x, y, a, f1, f2, f3
Integer :: b,c

!Assign numerical values:
x = 1.1; y = 2.5; a = -5.5; b = 10; c = 3;

!Add, subtract, multiply and divide some numbers:
f1 = (x+y)/y
f2 = (a*b) + (c-a)
f3 = (x-a)/(-a*y)

!Print results to terminal:
Print*,"x=",x
Print*,"y=",y
Print*,"a=",a
Print*,"b=",b
Print*,"c=",c
Print*,"(x+y)/y=",f1
Print*,"(a*b) + (c-a)=",f2
Print*,"(x-a)/(-a*y)=",f3

STOP;
END PROGRAM math
```

3.3 Variable declaration

Fortran is implicit language and all variables need to be declared. The compiler needs to know the names, types, and sizes of the variables used in the program in order to allocate memory and optimize the performance. All variables must be declared if the `IMPLICIT NONE` statement is used. Otherwise a type is implied by the first letter:

- a…h and o…z are `REAL`
- i,j,k,l,m,n are `INTEGER`

It is highly encouraged to explicitly declare all variables by using the Implicit None statement as shown previously in the sample program math. Explicit declaration is done with the statement:

```
type [, attributes] :: variable = [initial value]
```

3.3.1 Naming conventions

The rules for naming conventions for all variables in Fortran 90 are as follows:

- Names can be up to 31 characters long.
- Allowed symbols are letters a… z, numerals 0, 1, 2, …,9 and the underscore _.
- First character must be a letter.
- Variables are case insensitive.
- Cannot use reserved words (i.e. EXIT, EXP, SIN).

Table 3.1 shows some example valid and invalid variable names.

3.3.2 Data types

In your programs, you will use a variety of different types of variables such as integers, floating point numbers, strings of letters, etc. Table 3.2 summarizes supported Fortran 90 data types.

Integer constants are of the form 1234, real constants are of the form 1234.0 or 1.234E03. The range of integer values is limited, but the exact value depends on the compiler and machine used. Real*8 constants are of the form 1.234D03 (often refer to double precision), and complex numbers which are represented by an an ordered pair of Reals are of the form (3.14,–1E05). Characters are enclosed in quotes of the form 'ABab' or 'S'. The Character statement requires the length of the string which is written in a couple of ways:

```
Character (Len=10) :: position
Character (20) :: acceleration
```

Table 3.1. Variable names.

Valid variable names	Invalid variable names
x,Y,t,a	x*y, x+y
temp11	33y, 11temp
delta_x	delta y
bsquared	_one_two
really_Longname10	sin, log, exp

Table 3.2. Fortran 90 data types.

Data type	Description
`Integer`	Whole numbers
`Real`	Floating point numbers (~8 digits)
`Real*8`	Floating point numbers (~16 digits)
`Character`	Strings of characters
`Logical`	Two-valued Boolean variables
`Complex`	Complex number

In this example above, the variable 'position' can hold up to ten strings of characters. Note, that the keyword `Len=` is optional and can be omitted. Thus, the variable 'acceleration' can hold up to 20 strings of characters.

`Logical` constants can have only two values `.TRUE.` or `.FALSE.` (note the opening and closing periods). Sample syntax is provided below:

```
Real*8 :: grav=6.67E−11
Real :: pi=3.1415, rho, g = 9.81
Integer :: f = 30, mu, row, col
```

Fortran 90 attempts to make code more transferable and architecture independent by allowing the user to specify the length (in words) of the variable through the `KIND` attribute (note that this keyword can be omitted):

```
type([KIND=]kind_num) [, attributes] :: variable list
```

```
Real (Kind = 8) :: au = 1.5E11
Real (8) :: double_number−2.0D3
```

The problem is that a word is defined differently on different machines. On most architectures (such as the PC), for example, a `REAL` with eight words is double precision, and with four words is a single precision float. A good programmer will make one separate record (module) for each of the machines the program runs on, and make symbolic names for the kinds that will be universal, while changing the length depending on the machine architecture.

If you are using a 'constant' throughout your program, you may want to use the `Parameter` statement to define this. Once defined, the value cannot change throughout your program.

```
Real*8 :: G, pi
Parameter (G=6.67E-11)
Parameter (pi=acos(-1.0))
```

Generally, you do not have to necessarily define the values of certain variables at the moment of declaration. You may easily assign numerical values after you have declared the type of variable. This may be more convenient if you have many different variables or if the value of the variable will depend on some other external source. However, all declaration of variables and parameter statements must be done prior to using those variables.

3.4 Basic expressions

3.4.1 Arithmetic operators and expressions

The most important arithmetic operators are addition +, subtraction −, multiplication *, division /, and exponentiation **. For the following examples

```
A+B-C; A*(-B); A*B/C; Z**I
```

The order of evaluation is:

1. Parentheses (innermost first)
2. Exponentiation
3. Multiplication and division
4. Addition and subtraction

For example, the expression `A*B-C/D` is evaluated as if it were written as `(A*B)-(C/D)`. Where two operators have the same priority, the order of evaluation is left to right. For example, `A/B*C` is evaluated as `(A/B)*C` which is not equivalent to `A/(B*C)`, of course. It is important to notice that the result of an operation between floats is a float, between integers is an integer, and between floats and integers is a float. This means, for instance, that 1/5 gives the integer 0 while 1./5. gives 0.2 in single precision, and `1D0/5D0` gives 0.2 in double precision. In general if `I` is an integer (say 4), use `I.0` (4.0) to make it a floating point number.

3.4.2 Relational operators

Many type of programs require the user and the code to make choices. These choices will depend on certain conditions set by the code. If the conditions are met, certain outcomes will happen, otherwise other outcomes will happen. These logical conditions are categorized as:

- equal
- greater or equal to
- greater than
- less than or equal to

- less than
- not equal to
- not
- and
- or
- equivalent
- not equivalent

Logical operator Fortran 90 syntax for these conditions is described in table 3.3. For example, `(A >= B)` is true if the value of the variable `A` is greater than or equal to the value of `B`. Operators can be applied between mixed types, and as with arithmetic operators, all the arguments are converted to the highest type (integer → real → double → complex).

3.4.3 Logical expressions

You can combine relational expressions and other 'true–false' valued expressions and variables together to form logical expressions. This is done using the logical operators shown in table 3.3. Since some of you may not be familiar with Boolean logic, an overview of Boolean logic is provided in table 3.4 (T stands for true, F for false).

Table 3.3. Overview of relational operators.

Relational operator	Meaning
`==`	equal to
`>=`	greater than or equal to
`>`	greater than
`<=`	less than or equal to
`<`	less than
`/=`	not equal to
`.NOT.`	logical negation
`.AND.`	logical and
`.OR.`	logical inclusive or
`.EQV.`	logical equivalence
`.NEQV.`	logical non-equivalence (exclusive or)

Table 3.4. Overview of Boolean logic.

`x`	`y`	`.NOT.x`	`x .AND. y`	`x .OR. y`	`x.EQV. y`	`x .NEQV. y`
F	F	T	F	F	T	F
T	F	F	F	T	F	T
F	T	T	F	T	F	T
T	T	F	T	T	T	F

The only rule that you need to know (in case of doubt use excessive bracketing, if necessary) is that arithmetic operators take precedence over relational operators which take precedence over logical operators. Thus, these two lines are equivalent; however, it is usually safer to use the second line.

```
x == y .OR. b > pos .AND. acceleration < g
(x == y) .OR. (b > pos) .AND. (acceleration < g)
```

3.5 Input and output

3.5.1 The READ statement

Most programs require that the user input some data through the keyboard or that the program prints some result on the monitor. User input is achieved through the statement READ, with structure(s)

```
READ*, {input list}
READ ([UNIT=]unit type, [FMT=]format){ input list}
```

The unit type is a number, for example, a 5 for keyboard, and the format type is * for a format-free input. For example,

```
READ (UNIT=5, FMT=*) a, b, c
```

expects the user to input three numbers separated with commas.

3.5.2 The WRITE statement

Write to the screen (or printer) is done using the command WRITE, which can take the form:

```
PRINT *, {output list}
WRITE ([UNIT=unit type], [FMT=]format) { output list}
```

The unit type is again a number, usually 6 for the monitor. For example,

```
WRITE (6,*) 'The total is:', total
```

prints the message in the quotes and then prints the value of the value total.

3.5.3 The FORMAT specification

The format can be specified by the user in two different ways. First, the format can be a number giving the label (in columns 1–5 of the program) of the line in the program containing the call:

```
FORMAT(format sequence)
```

or the format can be a string containing the format sequence, with the syntax

```
FMT=('format sequence')
```

The format sequence is a beast of its own. It is a list of data descriptors and control functions, which specify what type the data that is being printed is and how it should be printed and specify things like new lines.

Data descriptors

Table 3.5 gives the data descriptors for various types of data. The capital letters in table 3.5 have specific meanings and are described below:

I – integers
F – a fixed-point floating number
E – a float in scientific (exponential) notation
G – either F or G, depending on the magnitude of the number
L – logical
A – character

with the lower-case letters meaning:

w – the total field width
m – the minimum number of digits
d – number of digits after the decimal point (precision)
e – number of digits in the exponent

For example,

```
WRITE(UNIT=*, FMT=10) 'The frequency is', f, ' Hz'
10 FORMAT(1X, A, F10.5, A)
```

Table 3.5. Data descriptors.

Data type	Data descriptors
Integer	I*w*, I*w*.*m*
Floating point	E *w*.*d*, E*w*.*d* E*e*, F*w*.*d*, G*w*.*d*, G*w*.*d* E*e*
Logical	L*w*
Character	A, A*w*

displays the value of the frequency variable `f` with a maximum of 15 digits, 5 decimal digits, an indent of 1 column in the beginning, and a suitable string message.

3.5.4 File input and output (I/O)

One of the areas in which Fortran 90 (besides number crunching) is better than most other programming languages is file input and output. In most programming languages, file does not refer just to hard disk files, but to any peripheral device that one can read from or write data to (keyboard, monitor, printer, disks). Therefore, file I/O is still done using the `READ` and `WRITE` commands (usually), but with a UNIT number that specifies the device or the file the data is being read from or written to.

OPEN
To assign specific UNIT numbers to certain files and open (or create) the file, use the `OPEN` statement:

```
OPEN(UNIT=Integer, FILE='filename', STATUS=literal)
```

The literal keyword can be '`NEW`' for new files, '`OLD`' for existing files, '`UNKNOWN`' if you are not sure or '`SCRATCH`' for temporary files. There is one more important keyword worth mentioning here. The keyword `ACCESS=access` which would go after the `STATUS` keyword is one of '`SEQUENTIAL`' or '`DIRECT`'. The distinction is very important, but beginners should use the default of sequential files, which are simply files in which the data are written and read from line-by-line.

CLOSE
After we are done using the file, it is highly recommended that the file be closed using:

```
CLOSE(UNIT=number, STATUS=status, ...)
```

where the unit is the number (must be an integer value) of the file, and the default is 'KEEP' to save the file or 'DELETE' to delete it.

3.6 Control structures

Control structures determine the program flow—the sequence of execution of the commands (other than the default ordering in the program file). In Fortran 90 these are the `IF` blocks and the `DO` loops.

3.6.1 IF-blocks

An `IF` block is a multi-branched control structure which takes the execution to different branches depending on the value of one or several condition statements. The general structure of `IF` blocks is as follows:

```
IF(First condition statement) THEN
    First sequence of commands
ELSE IF (Second condition statement) THEN
    Second sequence of commands
ELSE IF ...
...
ELSE
    Alternative sequence of commands
END IF
```

Any but the first IF can be omitted. The nth (ELSE) IF sequence of statements is evaluated if its condition statement is true and if none of the preceding conditions were true. In other words, once a condition statement is found true, its sequence of commands is evaluated and the IF-block is exited. The ELSE sequence of commands is evaluated if none of the condition statements are true. For example, to find the sign of a number (signum(x) = 1 for $x > 0$, −1 for $x < 0$, and 0 for $x = 0$) we can use the following construction:

```
IF (number < 0) THEN
    signum=-1
ELSE IF (number > 0) THEN
    signum=1
ELSE
    signum=0
END IF
```

Many IF statements are one-liners and more clarity and less typing are achieved using an abbreviated IF one-liner:

```
IF (Condition statement) Statement to be executed
```

which simply omits the THEN and END IF in the usual conditional IF-block. The following is a sample program which illustrates the If–Then–Else structure:

```
Program TryIf
write(*,100)
Read(*,*) X
```

```
  If (X < 0) Then
  write(*,*) X, 'is a negative number'
  elseif (X > 0) Then
  write(*,*) X, 'is a positive number'
  else
  write(*,*) 'X is 0'
  endif
 100 format('Please input a number: ',$)
  end program TryIf
```

3.6.2 DO loops

Loops are blocks of commands to be repeated as illustrated below:

```
Do i = start, finish, increment
   Executable Statements
End Do
```

where `i` must be an integer value and is called the control variable. The `increment` keyword is optional. If it is specifically not listed, the default increment is 1. For example,

```
Do i = 1, 10
   Print*, i**2
End Do
```

the loop variable `i` will start at `i = 1` and print `1*1 = 1`, then the loop variable will go to the next value determined by the increment, (in this case, the increment is 1) which is `i = 2` and print `2*2 = 4` and thus will continue to the value of 10.

Instead of incrementing of values of 1, what if the increment is some other non-integer value such as 0.5? This requires a little bit of thought. Having this increment be implemented correctly in the loop control structure is a very important part of iterating correctly. There are a few important concepts to think about here:

1. First, If we are going to increment in units of '0.5' we are going to have to somehow declare that, this is commonly known as a 'step-size'.
2. Second, since we are incrementing half a unit, we are going to need to double our iterations.
3. Lastly, we are going to need some expression that will allow us to print out the correct results.

The sample code below illustrates these three important concepts:

```
Program loops
 Implicit None

 Real :: dn, f
 Integer :: n

 dn=0.5

 Do n=1,20
 f=dn*n
 Print*, f**2
 End Do
 Stop
End Program loops
```

In the sample code above, `dn` is our step-size and the variable `f` is our function which is defined by our step-size times the loop variable `n`. Our last entry (f_{20} = (0.5)(20) = 10 = 10*10 = 100) just as before. All we have changed is the increment and thus the 'amount' of data generated.

You may be wondering, why did we apply these certain extra steps? Could we not simply just add our increment such as:

```
Do i = 1, 10, 0.5
   Executable Statements
End Do
```

and obtain the same results with fewer lines of code? The problem with the above example program is that it will only work for certain compilers which will not distinguish between `Reals` and `Integers` as loop control statements. As mentioned earlier in this chapter, good and efficient programming should always be compiler independent—that is the sample full program shown previously which we explicitly declared our 'step-size' will compile using any compiler, while if we used `Do i = 1, 10, 0.5` instead it would only work on certain compilers.

The `Do While` loop

You can loop over quantities with a condition as well by using the `Do While` statement:

```
Do While (Condition)
   Executable Statements
End Do
```

This loop will execute as long as the `condition` stays true. A sample program is provided for your reference

```
Program dowhile
 Implicit None

 Real :: none
 Integer :: counter

 counter = 0
Do While (counter < 10)
 counter = counter + 1
 Print*, counter
End Do
Stop; End Program dowhile
```

In this sample program, we are iterating the variable counter as long as the counter variable is less than 10. The first iteration, the counter is zero, thus the variable then equals counter = 0 + 1 = 1. The second iteration would be counter = 1 + 1 = 2 and so on and so forth as long as the condition of `counter < 10`. Once the the counter reaches a value which is equal or greater than 10, the program will stop and end.

3.6.3 Nested loops

Just as single loops described in the previous section, control structures with loops can be nested as well. The standard statements for nested loops are as follows:

```
Do i = 1, 10
   Do j = 1, 10
    Executable Statements
   End Do
End Do
```

Nested loops start ‘inner’ and go ‘outer’. For example

```
Do i = 1, 10
   Do j = 1, 10
    Print*, i*j
   End Do
End Do
```

the program is iterating the loop variable `j` then `i`. More specifically, our iterations will be as described by the expressions in equations 3.1 and 3.2.

$$\begin{array}{l} i\ = 1,\ j = 1\ \Rightarrow\ i \cdot j = 1 \\ i\ = 1,\ j = 2\ \Rightarrow\ i \cdot j = 2 \\ i\ = 1,\ j = 3\ \Rightarrow\ i \cdot j = 3 \\ \vdots\quad \vdots\ \vdots\ \vdots\ \vdots\ \vdots\ \vdots\ \vdots\ \vdots\ \vdots \\ i = 1,\ j = 10\ \Rightarrow\ i \cdot j = 10 \end{array} \tag{3.1}$$

$$\begin{array}{l} i\ = 2,\ j = 1\ \Rightarrow\ i \cdot j = 1 \\ i\ = 2,\ j = 2\ \Rightarrow\ i \cdot j = 2 \\ i\ = 2,\ j = 3\ \Rightarrow\ i \cdot j = 6 \\ \vdots\quad \vdots\ \vdots\ \vdots\ \vdots\ \vdots\ \vdots\ \vdots\ \vdots\ \vdots \\ i = 2,\ j = 10\ \Rightarrow\ i \cdot j = 20 \end{array} \tag{3.2}$$

3.7 Modular programming

3.7.1 Intrinsic functions

There is a library of intrinsic functions available to any Fortran program. These functions are invoked by using the function name followed by its parenthesized list of parameters:

```
function name({list of parameters})
```

A function is said to return a value based on the values passed to it through the arguments. Some of the more important intrinsic functions are `ABS`, `ACOS`, `COS`, `DOT_PRODUCT`, `EXP`, `INT`, `LEN`, `LOG`, `SIN`, and `SQRT`. See appendix A for a fuller list and details. Many intrinsic functions are generic in the sense that the functions may be used with different (but not mixed) data types as parameters to the function (e.g. `ABS`, `COS`, `MAX`, and `MIN`). Some functions, however, like the float-valued functions `SQRT`, `SIN`, and `LOG`, accept only certain types of arguments. For instance, `SQRT(4)` is invalid; one needs to type `SQRT(4.0)` instead. Some functions have different versions for different types of arguments (see appendix A), and the generic function calls these specialized functions depending on the types of the arguments. For example, to compute the square root of a given integer or any floating point number (even complex), we simply write `root=SQRT (1.0*number)`.

3.7.2 Intrinsic subroutines

Subroutines are very similar to functions, but there is an important distinction. Functions return one value and are not recommended to change the values of their

parameters. Subroutines do not return a value explicitly, but execute a well-defined group of statements (activity) and can freely change the values of their parameters. They should also be used when we wish to return more than one value. Subroutines are invoked using a CALL statement,

```
CALL name({list of arguments})
```

Some of the intrinsic subroutines include DATE_AND_TIME, MVBITS, RANDOM_NUMBER, RANDOM_SEED, and SYSTEM_CLOCK.

3.7.3 External functions

You can define your own functions, usually after the END of the program. The layout for functions is

```
type FUNCTION name({dummy arguments})
    local variable declaration
    name = expression
    body of function continued if need ...
END FUNCTION name
```

The type of the name identifies the type of the result that will be returned by the function. The result of the function is the value of the function that is assigned to the name of the function within the body of the function. The function therefore must contain at least one assignment of a value to the name of the function. The dummy arguments are a list of constants, variables and even procedures which are accessible from within the body of the function. When the function is used, it will have corresponding actual arguments that must be of the same type and length as the dummy arguments, but not necessarily the same names. No type checking is done during compilation or run time, and you will get very weird errors if the types do not match. All arguments are passed using 'call by reference' (like VAR arg in Pascal or & arg in C++). Changing the value of an argument in the subroutine changes the value of the corresponding variable in the calling program. For example, to define a function that computes the gravitational force between two bodies, we define a function NEWTON as follows:

```
REAL FUNCTION Newton(m1, m2, r)
    REAL (Kind=8) :: gamma=6.672E(−11), r
    REAL :: m1, m2
    Newton = −gamma*m1*m1/r**2
END Newton
```

A sample code illustrating the use of external functions is provided for your reference. In this program, the code reads in three real numbers and calculates the average. The user defined function ave returns a value to VAL.

```
PROGRAM Average

 IMPLICIT NONE
 REAL :: TEST1, TEST2, TEST3, ave, VAL
 PRINT*, 'Enter three numbers, separated by a comma'
 READ *, TEST1, TEST2, TEST3
 VAL = ave(TEST1,TEST2,TEST3)
 PRINT *, VAL
 STOP
 END PROGRAM Average

FUNCTION ave(X,Y,Z)
 IMPLICIT NONE
 REAL :: X, Y, Z, ave

 ave = (X + Y + Z) / 3.0
RETURN
END FUNCTION ave
```

3.7.4 External subroutines

The structure of a subroutine is

```
SUBROUTINE name({dummy arguments})
   local variable declaration
   body of subroutine ...
END SUBROUTINE name
```

Subroutines are accessed by using the CALL statement, and are in most respects similar to functions. When a subroutine is called, the dummy arguments in the subroutine become alias names for the actual arguments in the calling statement, i.e. they represent the same physical location in memory. Thus, if the dummy arguments are modified within the subroutine, then so are the actual arguments in the calling statement.

The sample program provided earlier illustrating functions can be easily modified to include an external subroutine. See below for the full program which includes an external subroutine.

```
PROGRAM Average

 IMPLICIT NONE
 REAL :: TEST1, TEST2, TEST3, VAL
 PRINT*, 'Enter three numbers, separated by a comma'
 READ*, TEST1, TEST2, TEST3
 CALL AVG(TEST1,TEST2,TEST3, VAL)
 PRINT *, VAL
 STOP
 END PROGRAM Average
SUBROUTINE AVG(X,Y,Z,V)
 IMPLICIT NONE
 REAL :: X, Y, Z, V
 V = (X + Y + Z) / 3.0
RETURN
END SUBROUTINE AVG
```

All well-defined functions should have a single-point entry and a single-point exit, but in some situations there may be a need to terminate a procedure (e.g. in case of error). This is done by simply calling `RETURN`, which stops the execution of the function and returns control to the calling routine.

After the function is finished, the values of the local variables are lost. If they are to be used in a later call of the function, this should be made clear by saving the values of these between function calls (static allocation):

```
SAVE {of values to be saved}
```

When a function is passed on as an argument to another function, and in some circumstances when we wish to link a named block data subprogram into the final executable, the function has to be declared as either intrinsic or external, via the following statements:

```
INTERNAL {list of function names}
EXTERNAL {list of function names}
```

3.7.5 Program units

Each program consists of program units, called *scoping units*. These can be the main program, subprograms, procedures, etc. The important thing to notice is that each scoping unit is independent in terms of its variable space, so that it is good to try to put well-defined and more or less isolated groups of statements into separates units. This way, you can invoke that unit from various programs, without worrying about possible variable conflicts.

Any unit that is placed (defined) within another unit is a *sub-unit* and has access to all the variables in the unit it is a part of. Therefore, procedures that interact with the main program variables should be placed inside the PROGRAM, using the CONTAINS command. These are *internal procedures*, as contrasted to *external procedures*, which are defined outside the PROGRAM unit and have their own variable workspace. The communication, or *interface*, between the main program or subprogram and the external procedure is done through the argument list. Data are copied from actual arguments to dummy arguments upon the invocation of the procedure (this is not always true), and from dummy argument to actual argument upon exiting the procedure.

Another way of sharing data between procedures is the usage of modules, which are new units introduced in Fortran 90, facilitating global variables and procedures. We now describe in more detail some of the new features of the Fortan 90 standard.

3.7.6 Internal procedures

Internal procedures should be placed after the CONTAINS command. They can be a part of any program unit, like the main program, a subprogram or a procedure (usually no more than 2–3 levels of nesting). For example, here is how to make a procedure that quickly zeros all the integer counters that we use in the main program:

```
PROGRAM counters
 INTEGER :: i, j, k, l, m, n
 CALL Zero_counters ! Zero all the counters
 CONTAINS

SUBROUTINE Zero counters
 i = 0; j = 0; k = 0; l = 0; m = 0; n = 0;
END SUBROUTINE Zero_counters

END PROGRAM counters
```

3.7.7 External procedures

As already explained, external procedures are usually placed outside the main program (in a separate file, a module, the same file, etc). When the compiler compiles the program, it can detect a great deal more errors if it is told what the procedure's interface (argument list) is. This is done with the INTERFACE block, which contains the declarations of the external functions (it can be placed inside the main program or in a separate module). Also, the compiler can be told what the intention of the arguments of a subroutine is, using the INTENT data attribute with one of the following arguments:

- IN for arguments that should not be altered in the subroutine, but only pass data to the procedure.
- OUT for arguments that need to be assigned values in the subroutine.
- INOUT for arguments that pass data both in and out of the procedures.

For example, here is an interface to a subroutine that returns the time in seconds given the time in hours, minutes, and seconds:

```
INTERFACE
SUBROUTINE convert_time(hour,minute,second,time)
    INTEGER, INTENT(IN):: hour, minute, second
    INTEGER, INTENT(OUT):: time
END SUBROUTINE convert_time
END INTERFACE
```

3.7.8 Modules

Modules are a new program unit in Fortran 90. They are mostly used to enclose data (global variables) and `INTERFACE` declarations of functions that several other program units need to share. One can also put full procedure bodies inside a module (so called module programs). The modules are defined as

```
MODULE name
    Global variable declarations
    INTERFACE blocks
    ...
END MODULE name
```

If we want a program unit to have access to the variables and procedures defined in the module, we use the `USE` command. If we have lots of procedure interfaces in the module (for example, a subroutine library) but would want to use only a few of these procedures, we specify this with the `ONLY` attribute (this is also a way to make some global variables not accessible to all program units and is good programming practice):

```
USE module name [, ONLY: {procedure names}]
```

For example, we can store the total number of floating-point operations (flops) in a program in a module,

```
MODULE flops
    INTEGER(KIND=8) :: flops_count
END MODULE flops
```

and then update this number from any program unit that contains the statement

```
USE flops
```

3.8 Arrays

Arrays are collections of data of the same type. They are the most important data type in scientific computing where we usually deal with a great number of data (points).

3.8.1 Declaration of arrays

Arrays are declared in the same way as ordinary variables, with lower and upper bounds along with important attributes

```
type [,attributes] array({Lower bound:Upper bound})
```

These attributes are:

- DIMENSION(Lower bound:Upper bound) which enables the dimension to be specified, and the lower and upper count can now be actual workspace variables, thus enabling the existence of *automatic arrays* in procedures, which come to exist with different dimensions each time the procedure is called. Also, it is very important to note that the dimensions do not have to be specified at compile time, in which case they are substituted with the ellipse symbol. If the dimensions of an array argument to a procedure are not specified, the procedure must have an INTERFACE block in the calling unit, so that the compiler knows that the array has to be passed along with its dimensions.
- ALLOCATABLE which identifies the array as allocatable.
- TARGET which identifies the array as a target for pointers.

For example, to declare an array used to store your income for several years and all the months in each year, one can use

```
REAL, DIMENSION(12,1995:1999) :: income
```

3.8.2 Vectors

Vectors are one-dimensional arrays which are mainly used to access sub-arrays of a given array (see next subsection). It differs from an array only in the way it is assigned a value by using constructors:

```
vector = (/{lower:upper:step],.../})
```

where the list may be a list of values of the appropriate type, namely, a variable expression, array expression, implied DO loop or any combination of these. Here is an example of a vector that has the value *vector* = (1, 3, 5, 7, 9, 6, 2, 4,,8,16):

```
INTEGER :: vector(6)=(/1:9:2,6, (2**i,i = 1,4)/)
```

3.8.3 Using arrays

Arrays in Fortran 90 can be accessed on an element-by-element basis, in addition to access of whole sections by the following construct:

```
array({start:end:step})

array(vector)
```

Any of the starting, ending, and step values for the indices can be omitted, so long as there is one ellipse. Examples are illustrated in the following:

```
REAL :: array(50,50), vector(3)=(/1 7 37/)

!This accesses the section of the array from rows 1--20
!and columns 5--10, returning an array of size [20,5]
!WRITE(*,*) array(1:20,5:10)

!This accesses all the even-numbered rows
WRITE(*,*) array(2:50:2,:)

!And this accesses the elements at the intersection of
!rows 1, 7 and 37 with columns 1, 7 and 37
WRITE(*,*) array(vector,vector)
```

3.8.4 Array operations

A very important new feature in Fortran 90 is the idea of array–array operations between *conformable arrays*. Two arrays are said to be conformable if they have the same size (not necessarily the same row or column numbering), or if one of the two arrays is a scalar. An operation between two conformed arrays is carried between the elements of the two arrays individually. For example,

```
REAL, DIMENSION(20,20) :: A, B, C
C=A+B
```

assigns to each element in `C` the value of the sum of the corresponding elements of `A` and `B`. Thus, you may say that this is equivalent to a `DO` loop like

```
DO i=1,20
   DO j=1,20
    C(i,j)=A(i,j)+B(i,j)
   END DO
END DO
```

But this is not quite true, and it is important to understand the difference. The matrix statement `C = A+B` does not span in time—it spans in space. In other words, as far as summing two matrices goes, the order in which the summation is done is arbitrary. On the other hand, the double `DO`-loop structure spans in time, not in space. Therefore, the double loop is an unnatural translation of the matrix statement. The reason that nested looping is the traditional way of performing matrix operations is that until recently most computer people used so-called von Neuman sequential machines, in which there is a single processor that does things in a time-ordered fashion. However, today the concept of parallel, or multi-processor machines is an essential one, so that nested do loops should be avoided whenever an equivalent array–array (vectorized) statement exists. When the compiler sees a statement `C=A+B` for matrices, it automatically optimizes the process for the specific machine using the fact that the operation is not sequential. How it does that is fortunately not our concern. It may use several processors to do that if the machine has more than one, it may access the elements column- or row-wise, use pointers (those that are more experienced know that one of the fastest operations is the incrementation of a pointer, not an integer, by one, so that an array can be accessed most quickly with a sweeping pointer), etc. The lesson from all this is that you should always try to formulate your code in vectorized statements.

3.8.5 Elemental functions

To continue the previous discussion, let us assume we need to take the sine of all the elements of the array `A`. Again, this is not a time-ordered operation, and Fortran 90 offers a very effective way of doing this. We can simply use `SIN(A)` to perform this operation, because `SIN`, like most other numerical functions in Fortran 90, is an elemental procedure—acting on each element of an array. Thus a statement like

```
C(5:10,:) = SIN(A(2:10:2,:)) + 2.0 * B(5:10,:)
```

performs the very complicated matrix operation of assigning the fifth to tenth rows of `C` the sum of the sine of the elements of `A` in the even rows of `A` up to the 10th row and the doubled value of the elements of the matrix `B` in columns 5–10.

3.8.6 The WHERE statement

The `WHERE` construct is a very useful control structure for performing an operation only on certain elements of an array that satisfy a given condition. The structure is as follows:

```
WHERE (array condition)
   Body of structure
END WHERE
```

For example, the following operation

```
DO j = 1,3
   DO i = 1,4
    IF(array(i,j) > 0) array(i,j) = LOG(array(i,j))
   END DO
END DO
```

can be done much more cleanly using the WHERE construct:

```
WHERE (array > 0) array = LOG(array)
```

Another example is

```
REAL, DIMENSION(20,20) :: A, B, C
WHERE (B>0.0)
    C=A/B
END WHERE
```

which assigns each element of the matrix C the quotient of the corresponding elements of matrices A and B, so long as the element of B is non-zero. It should be noted again that the WHERE construct is not equivalent to a DO loop with a nested IF statement, since a WHERE loop is not time-ordered and can be performed in parallel. A major limitation of Fortran 90 is that WHERE constructs cannot be nested. This is corrected in Fortran 95, where WHERE and FORALL structures can be nested at any level.

The last example full program shown below assigns given values to a matrix, array. The values are either 1 or -9.999 999, depending on whether eval has the value 'true' of 'false':

```
PROGRAM WHERE_Example

IMPLICIT NONE
REAL, DIMENSION(2,3) :: array = (/0, -1, 2, -3, 4, -5/)
LOGICAL,   DIMENSION(2,3)::eval=(/.true.,.false.,.true.,
&
.true.,.false.,.false./)

WHERE (eval)
array = 1
ELSEWHERE
array = -9.999 999
END WHERE
```

```
WRITE (*,'(3F15.6)') array

END PROGRAM WHERE_Example
```

The result is

```
1.000000 -9.999999 1.000000
1.000000 -9.000000 -9.999999
```

3.8.7 FORALL (Fortran 95)

The FORALL structure is essentially a more general version of the WHERE construct. It may also be considered as a space-spanning equivalent to nested DO loops that perform complicated operations whose time execution must be arbitrary. The format of the FORALL statement is

```
FORALL (index=lower bound:upper bound:increment)
statement
```

Two full sample programs are shown below for your reference:

```
program where_construct
implicit none

integer, dimension(5) :: a = (/ 1, 2, 3, 4, 5 /)
integer :: i

forall(i = 1:4, a(i) >2) a(i) = 0

write(*, *) a

end program where_construct
```

The result is

```
 1 2 0 0 5
```

```
program where_construct
implicit none

integer, dimension(5) :: a = (/ 1, 2, 3, 4, 5 /)
```

```
integer :: i

forall(i = 1:5:2) a(i) = -1.

write(*, *) a
end program where_construct
```

The result is

```
 -1 2 -1 4 -1
```

3.8.8 Array intrinsic functions

Fortran 90 contains a lot of new intrinsic functions for arrays. It is beyond the scope of this text to explain all them in detail. However, here we have just a brief list of some of them and what they do, so you know what to look for when you need them:

- `SIZE(array,[dimension])` is probably the most important inquiry function for array which finds the rank of an array along the specified dimension (or total number of elements). Use `SHAPE(array)` to obtain the shape of the array as an array to integer ranks.
- `SUM(array,[dimension])` and `PRODUCT(array,[dimension])` return the sum or product of the elements of an array along the specified dimension.
- `MINVAL(array,[dimension])` and `MAXVAL(array,[dimension])` give the minimum and maximum value of an array along the specified dimension.
- `MATMUL(first array, second array)` is a very important function that returns the matrix product of two arrays (optimized for parallelization when possible).
- `DOT_PRODUCT(first vector, second vector)` finds the dot product of two vectors.
- `TRANSPOSE(matrix)` gives the transpose of a two-dimensional matrix.

3.8.9 Allocatable arrays

An essential improvement in Fortran 90 is memory management and the introduction of allocatable arrays, specified with the `ALLOCATABLE` attribute. These arrays do not have a specified dimension and are allocated or deallocated using the following commands

```
ALLOCATE(array({dimensions}))
DEALLOCATE(array)
```

For example,

```
REAL, DIMENSION(:,:), ALLOCATABLE :: array
ALLOCATE(array(10,40))
EALLOCATE(array)
```

first reserves memory for a 10 by 40 array, and then frees the memory to the memory pool.

A full sample program illustrating dynamic memory allocation is provided below. In this program, you can create a one-dimensional vector of size `dim_size`. The program user then will fill in the particular entries of the vector and the program will print out the size of the vector and its entries.

```
Program Testalloc
 Implicit None

 Integer :: i, dim_size
 Integer, Allocatable,Dimension(:):: vec

 Print *,'Enter the number of elements in the vector:'
 Read *,dim_size

 Allocate(vec(dim_size))
 Print *,'The size of your vector is:',dim_size
 Print*,"******"
 Print*,'Enter each entry of your vector:'

 Do i=1,dim_size
 Read *,vec(i)
 End Do

 Print *,'This is your vector'

 Do i=1,dim_size
 Print *,vec(i)
 End do

Deallocate(vec)
End Program Testalloc
```

3.8.10 Pointers

A pointer can be understood as a variable that points to a memory location (or locations). In particular, a pointer can point to a block of memory that we want to

use to store data, or, more importantly, it can point to the memory location of another variable, called the target of the pointer. A pointer is assigned a target via

```
pointer=>target variable of array
```

When a pointer is declared, it is born in an undefined status, when it is assigned a target it becomes associated, and to avoid inadvertent misuse we can make it disassociated with

```
NULLIFY(pointer)
```

For example,

```
REAL, DIMENSION(:), POINTER :: one_row
REAL, DIMENSION(50,50), TARGET :: array
one_row=>array(5,:)
NULLIFY(one_row)
```

associates the pointer to the fifth row of the array. This is a useful construction because it is more efficient and easy to manipulate, than, say, keeping in mind the number 5 to know which row we want to reference.

References

[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

Omair Zubairi and Fridolin Weber

Chapter 4

Numerical techniques

4.1 Curve fitting—method of least squares

The method of least squares assumes that the best-fit curve of a given type is the curve that minimizes the sum of the deviations squared from a given set of data. Let us suppose that the data points are $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, where x_i is the independent variable and y_i is the dependent variable and $i = 1, \ldots, n$, where N denotes the number of data points. The fitting curve $f(x)$ has the deviation (error) d_i from each data point y_i, that is, $d_1 = y_1 - f(x_1)$, $d_2 = y_2 - f(x_2)$, …, $d_N = y_N - f(x_N)$. According to the method of least squares, the best fitting curve is determined by the condition that $\Pi \equiv d_1^2 + d_2^2 + \cdots + d_N^2$ is a minimum. Mathematically, this can be expressed as

$$\Pi \equiv \sum_{i=1}^{N} d_i^2 = \sum_{i=1}^{N} (y_i - f(x_i))^2 = \text{minimum}, \tag{4.1}$$

which leads to a system of coupled linear equations for the unknown coefficients contained in the ansatz for $f(x)$. The simplest choices for $f(x)$ are a straight line, as shown in figure 4.1, or a parabola. Both cases will be discussed next.

4.1.1 The linear least-squares approximation

The linear least-squares method uses a straight line, $y = a + bx$, to approximate a given set of data, $(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$, where $N \geqslant 2$. From (4.1) it then follows that the best-fit straight line $f(x)$ has the least-squares error

$$\Pi \equiv \sum_{i=1}^{N} (y_i - f(x_i))^2 = \sum_{i=1}^{N} (y_i - (a + bx_i))^2, \tag{4.2}$$

which is to be minimized. Note that here a and b are unknown coefficients while all the data points x_i and y_i are given. The coefficients are found by minimizing Π of

doi:10.1088/978-1-6817-4896-2ch4 © Morgan & Claypool Publishers 2018

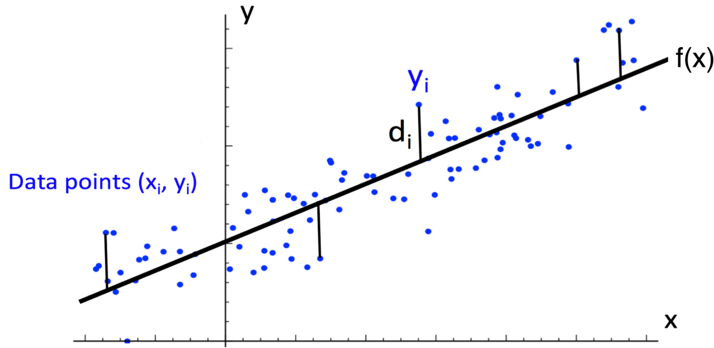


Figure 4.1. Experimental data points (x_i, y_i) fitted by a straight line $f(x)$. The quantity d_i defines the deviation of each data point y_i from the best-fit straight line $f(x_i)$ at each data point i, that is, $d_i = y_i - f(x_i)$.

(4.2), which leads to the conditions $\partial\Pi/\partial a = 0$ and $\partial\Pi/\partial b = 0$. From (4.2), these partial derivatives are given by

$$\frac{\partial\Pi}{\partial a} = 2\sum_{i=1}^{N}\left(y_i - (a + bx_i)\right) = 0, \tag{4.3}$$

$$\frac{\partial\Pi}{\partial b} = 2\sum_{i=1}^{N} x_i\left(y_i - (a + bx_i)\right) = 0. \tag{4.4}$$

Expanding equations (4.3) and (4.4), we obtain the following set of coupled equations (linear in a and b),

$$\sum_{i=1}^{N} y_i = a\sum_{i=1}^{N} 1 + b\sum_{i=1}^{N} x_i, \tag{4.5}$$

$$\sum_{i=1}^{N} x_i\, y_i = a\sum_{i=1}^{N} x_i + b\sum_{i=1}^{N} x_i^2. \tag{4.6}$$

Solving for a and b leads to

$$a = \frac{\left(\sum_{i=1}^{N} y_i\right)\left(\sum_{i=1}^{N} x_i^2\right) - \left(\sum_{i=1}^{N} x_i\right)\left(\sum_{i=1}^{N} x_i\, y_i\right)}{N\left(\sum_{i=1}^{N} x_i^2\right) - \left(\sum_{i=1}^{N} x_i\right)^2}, \tag{4.7}$$

$$b = \frac{\left(N\sum_{i=1}^{N} x_i\, y_i\right) - \left(\sum_{i=1}^{N} x_i\right)\left(\sum_{i=1}^{N} y_i\right)}{n\left(\sum_{i=1}^{N} x_i^2\right) - \left(\sum_{i=1}^{N} x_i\right)^2}, \tag{4.8}$$

which are easy to evaluate numerically.

4.1.2 The quadratic least-squares approximation

The quadratic least-squares method uses a second-degree polynomial (i.e. a parabola) $y = a + bx + cx^2$ to approximate a given set of data, (x_1, y_1), (x_2, y_2), …, (x_N, y_N), where $N \geqslant 3$. From (4.1) it now follows that the best-fit polynomial $f(x)$ has the least-squares error

$$\Pi \equiv \sum_{i=1}^{N} \left(y_i - f(x_i)\right)^2 = \sum_{i=1}^{N} \left(y_i - \left(a + bx_i + cx_i^2\right)\right)^2 = \text{minimum}. \tag{4.9}$$

For a parabola, a, b, and c are the unknown coefficients and all x_i and y_i data are given as for the straight-line case. To find the three unknown coefficients, we need to evaluates the three first-order derivatives

$$\frac{\partial \Pi}{\partial a} = 2\sum_{i=1}^{N} \left(y_i - \left(a + bx_i + cx_i^2\right)\right) = 0, \tag{4.10}$$

$$\frac{\partial \Pi}{\partial b} = 2\sum_{i=1}^{N} x_i \left(y_i - \left(a + bx_i + cx_i^2\right)\right) = 0, \tag{4.11}$$

$$\frac{\partial \Pi}{\partial c} = 2\sum_{i=1}^{N} x_i^2 \left(y_i - \left(a + bx_i + cx_i^2\right)\right) = 0. \tag{4.12}$$

Expanding equations (4.10) through (4.12), we have

$$\sum_{i=1}^{N} y_i = a\sum_{i=1}^{N} 1 + b\sum_{i=1}^{N} x_i + c\sum_{i=1}^{N} x_i^2, \tag{4.13}$$

$$\sum_{i=1}^{N} x_i\, y_i = a\sum_{i=1}^{N} x_i + b\sum_{i=1}^{N} x_i^2 + c\sum_{i=1}^{N} x_i^3, \tag{4.14}$$

$$\sum_{i=1}^{N} x_i^2\, y_i = a\sum_{i=1}^{N} x_i^2 + b\sum_{i=1}^{N} x_i^3 + c\sum_{i=1}^{N} x_i^4. \tag{4.15}$$

The unknown coefficients a, b, and c are thus given as solutions of the three coupled, linear equations (4.13)–(4.15).

4.2 Numerical differentiation

Derivatives of smooth, well-behaved functions can be approximated in several ways. The most basic ones are discussed in this section. We begin with the Taylor series expansion of a function $f(x)$,

$$f(x + \Delta x) = f(x) + f'(x)\Delta x + \frac{f''(x)}{2!}\Delta x^2 + \frac{f'''(x)}{3!}\Delta x^3 + \cdots, \tag{4.16}$$

where $\Delta x \ll x$. Evaluating the Taylor expansion (4.16) at $x + \Delta x$ and $x - \Delta x$ leads to

$$f(x + \Delta x) \approx f(x) + f'(x)\Delta x + \frac{f''(x)}{2!}\Delta x^2 + \frac{f'''(x)}{3!}\Delta x^3, \tag{4.17}$$

$$f(x - \Delta x) \approx f(x) - f'(x)\Delta x + \frac{f''(x)}{2!}\Delta x^2 - \frac{f'''(x)}{3!}\Delta x^3. \tag{4.18}$$

Subtracting (4.18) from (4.17) gives

$$f(x + \Delta x) - f(x - \Delta x) \approx 2f'(x)\Delta x + 2\frac{f'''(x)}{3!}\Delta x^3, \tag{4.19}$$

so that

$$f'(x) \approx \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x}, \tag{4.20}$$

which is known as leap-frog (central difference) differentiation. A less accurate way of numerical differentiation is the so-called Euler forward differentiation, which follows from (4.17) as

$$f'(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x}. \tag{4.21}$$

Finally, we mention Euler backward differentiation, which follows from (4.18) as

$$f'(x) \approx \frac{f(x) - f(x - \Delta x)}{\Delta x}. \tag{4.22}$$

A frequently used expression to compute the second-order derivative of a function $f(x)$ is obtained by adding (4.17) and (4.18) together. This leads to the three-point rule formula

$$f''(x) \approx \frac{f(x + \Delta x) - 2f(x) + f(x - \Delta x)}{\Delta x^2}. \tag{4.23}$$

Similarly, the five-point rule formula for the second derivative of $f(x)$ is given by

$$f''(x) \approx \frac{-f(x + 2h) + 16f(x + h) - 30f(x) + 16f(x - h) - f(x - 2h)}{12h^2}. \tag{4.24}$$

4.3 Numerical integration

There are several excellent methods that can be used to numerically compute the value w of a definite integral of the form

$$w = \int_a^b f(x)dx. \tag{4.25}$$

Here $f(x)$ is a smooth, well-behaved function defined for x-values in the range $a \leqslant x \leqslant b$, as illustrated in figure 4.2. A large class of numerical integration (quadrature) schemes can be derived by constructing interpolating functions which are easy to integrate.

4.3.1 The trapezoidal rule

In the simplest case, the functions used to interpolate $f(x)$ are polynomials of degree one, that is, linear functions, which are used to approximate the area under a function $f(x)$ for x-values between x_i and x_{i+1} by N trapezoids, as shown in figure 4.3. Here N is an integer which can be even or odd. Defining

$$x_k = a + kh, \quad \text{where } h = (b - a)/N, \quad (k = 0, 1, \ldots, N) \tag{4.26}$$

shows that the difference between two successive grid points on the x-axis is given by $x_{i+1} - x_i = h$. Since the area of each individual trapezoid is $hf(x_k) + h(f(x_{k+1}) - f(x_k))/2$, the total area covered by $f(x)$ follows as

$$\int_a^b f(x)dx \approx \sum_{k=0}^{N-1} hf(x_k) + \frac{1}{2}\sum_{k=0}^{N-1} h(f(x_{k+1}) - f(x_k)). \tag{4.27}$$

This expression can be written in the more compact form

$$\int_a^b f(x)dx \approx \sum_{k=0}^{N-1} hf_k + \frac{1}{2}\sum_{k=0}^{N-1} h\big(f_{k+1} - f_k\big), \tag{4.28}$$

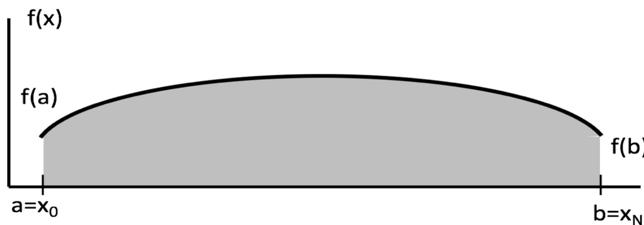


Figure 4.2. Graphical illustration of the value w (shaded area) of the integral of (4.25).

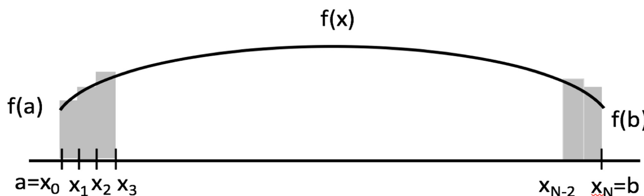


Figure 4.3. The the trapezoidal rule, the area covered by $f(x)$ is approximated by a sequence of vertical trapezoids of width $x_{i+1} - x_i$.

where $f_k \equiv f(x_k)$ and $f_{k+1} \equiv f(x_{k+1})$. Next, we re-write the summations in (4.28) as follows,

$$\sum_{k=0}^{N-1} hf_k + \frac{1}{2}\sum_{k=0}^{N-1} h(f_{k+1} - f_k) = hf_0 + h\sum_{k=1}^{N-1} f_k + \frac{h}{2}\sum_{k=0}^{N-1} f_{k+1} - \frac{h}{2}\sum_{k=0}^{N-1} f_k, \tag{4.29}$$

and furthermore

$$\sum_{k=0}^{N-1} f_{k+1} = \sum_{k=1}^{N-1} f_k + f_N, \tag{4.30}$$

$$\sum_{k=0}^{N-1} f_k = f_0 + \sum_{k=1}^{N-1} f_k. \tag{4.31}$$

This allows us to write the expression for the integral in (4.28) as

$$\int_a^b f(x)dx \approx \frac{h}{2}\left(f_a + 2\sum_{k=1}^{N-1} f_k + f_b\right), \tag{4.32}$$

with $f_a \equiv f(a)$, $f_b \equiv f(b)$, and x_k and h defined in (4.26). Equation (4.32) is known as the trapezoidal rule. The sample code below illustrates how the trapezoidal rule can be implemented in a numerical code:

```
h=(b–a)/FLOAT(N)
sum=0.
DO k=1,N–1
     x_k=a+h*FLOAT(k)
     sum=sum+f(x_k)
END DO
     trap = h * (f(a)+f(b)+2.*sum) / 2.
```

4.3.2 Simpson's rule

The second numerical integration technique considered in this text is Simpson's rule, which is often more accurate than the trapezoidal rule, since it uses a second-order polynomial (i.e. a parabola) $P(x) = Ax^2 + Bx + C$ rather than a linear function to approximate $f(x)$ between grid points x_i and x_{i+2}. The situation is illustrated graphically in figure 4.4. The quantities A, B, and C are constants which can be determined by integrating $P(x)$ from x_1 to x_{i+2},

$$\int_{x_i}^{x_{i+2}} P(x)dx = \frac{x_{i+2} - x_i}{3}\left(A(x_{i+2}^2 + x_{i+2}x_i + x_i^2) + \frac{3}{2}B(x_{i+2} + x_i) + 3C\right). \tag{4.33}$$

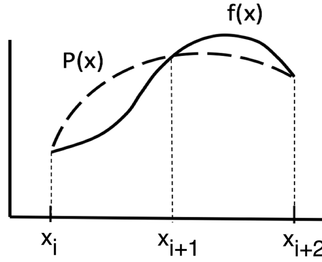


Figure 4.4. Simpson's rule can be derived by approximating the integrand $f(x)$ by a quadratic interpolant $P(x)$ between grid points x_i and x_{i+2}.

With a little bit of algebra, equation (4.33) can be written as

$$\int_{x_i}^{x_{i+2}} P(x)dx = \frac{x_{i+2} - x_i}{6}\left(P(x_i) + 4P\left(\frac{x_i + x_{i+2}}{2}\right) + P(x_{i+2})\right). \tag{4.34}$$

Now let us suppose that the integration interval $[a, b]$ is split up into N sub-intervals, where N is an even integer. Then, by applying the rule derived in (4.33) to integrate $f(x)$ over the first two intervals a, x_2 (see figure 4.3) leads to $w_2 \equiv (f_0 + 4f_1 + f_2)/3$, where h denotes the width of the interval, that is, $h = x_2 - a$. Repeating this step for all subsequent pairs of adjacent intervals leads for the integral to

$$\begin{aligned}\int_a^b f(x)dx \approx\ & \frac{h}{3}(f_0 + 4f_1 + f_2) + \frac{h}{3}(f_2 + 4f_3 + f_4) + \frac{h}{3}(f_4 + 4f_5 + f_6) \\ & + \cdots + \frac{h}{3}(f_{N-2} + 4f_{N-1} + f_N),\end{aligned} \tag{4.35}$$

which can be written as

$$\begin{aligned}\int_a^b f(x)dx \approx & \frac{h}{3}(f_0 + f_N) + \frac{h}{3}(f_0 + 2f_2 + 2f_4 + \cdots + 2f_{N-2} + f_N) \\ & + \frac{h}{3}(4f_1 + 4f_3 + 4f_6 + \cdots + 4f_{N-1}).\end{aligned} \tag{4.36}$$

One sees that the f_k values are multiplied by a factor of 2 or 4 depending on whether the grid point is even or odd, respectively. This is different for the trapezoidal rule of (4.32), where all f_k values are multiplied by the same weight factor. In summary, Simpson's rule is given by

$$\int_a^b f(x)dx \approx \frac{h}{3}\left(f_a + 2\sum_{\substack{k=2 \\ \uparrow \\ \text{even}}}^{N-2} f_k + 4\sum_{\substack{k=1 \\ \uparrow \\ \text{odd}}}^{N-1} f_k + f_b\right), \tag{4.37}$$

where $f_a \equiv f_0$ and $f_b \equiv f_N$. This formula can also be written as

$$\int_a^b f(x)dx \approx \frac{h}{3}\left(f_0 + \sum_{k=1}^{N-1}\left(2\delta_{(-1)^k,+1} + 4\delta_{(-1)^k,-1}\right)f_k + f_N\right), \tag{4.38}$$

where the Kronecker delta has been used to specify the corresponding weight factor. The Kronecker $\delta_{k,l}$ has a value of 1 if $k = l$ and 0 if $k \neq l$. As for the trapezoidal rule (see (4.26)), the grid points are given by $x_k = a + kh$, where $k = 0, 1, \ldots, N-1, N$, and $h = (b-a)/N$. The sample code below shows how Simpson's rule can be implemented in a numerical code:

```
h=(b-a)/FLOAT(N)
sum=0.
DO k=1,N-1
     x_k=a+h*FLOAT(k)
     weight=4.
     sign=(-1)**k
     if(sign > 0) weight=2.
     sum=sum+weight*f(x_k)
END DO
simp = h * (f(a)+f(b)+sum) / 3.
```

4.4 Matrix operations

In this section we briefly review key matrix definitions and basic matrix operations. These are transposition, multiplication of two matrices, and multiplication of a matrix with a vector.

Transposition: Let A be a $N \times N$ matrix, such as

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \cdots & \cdots & \cdots & \cdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{pmatrix}. \tag{4.39}$$

The transpose, A^T, of A is obtained by mapping $a_{jk} \to a_{kj}$ for all j and k values from $1, \ldots, N$, that is,

$$A^T = \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{N1} \\ a_{12} & a_{22} & \cdots & a_{N2} \\ \cdots & \cdots & \cdots & \cdots \\ a_{1N} & a_{2N} & \cdots & a_{NN} \end{pmatrix} \tag{4.40}$$

Multiplication of two matrices: Let A and B be two $N \times N$ matrices. Then their product $A \times B$ is given by the $N \times N$ matrix

$$C \equiv \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \cdots & \cdots & \cdots & \cdots \\ a_{41} & a_{42} & a_{43} & a_{NN} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1N} \\ b_{21} & b_{22} & b_{23} & b_{2N} \\ \cdots & \cdots & \cdots & b_{3N} \\ b_{41} & b_{42} & b_{43} & b_{NN} \end{pmatrix}, \tag{4.41}$$

with the individual matrix elements c_{jk} of C given by

$$c_{jk} = \sum_{l=1}^{N} a_{jl}\, b_{lk}. \tag{4.42}$$

Multiplication of a matrix with a vector: Let A be a $N \times N$ matrix, and x be a column vector with N elements. Then $A\, x$ leads to a new column vector y, according to

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \cdots & \cdots & \cdots & \cdots \\ a_{N1} & a_{N2} & a_{N3} & a_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_N \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \cdots \\ y_N \end{pmatrix}. \tag{4.43}$$

Algebraically, this system of equations can be written as

$$\sum_{j=1}^{N} a_{ij}\, x_j = y_i, \quad \text{for} \quad i = 1, \ldots, N. \tag{4.44}$$

The Jacobi iterative method can be used to solve certain systems of linear equations. To demonstrate the method, let us assume that the linear equation has the form $A\, x = b$ so that

$$\sum_{j=1}^{N} a_{ij}\, x_j = a_{ii}\, x_i + \sum_{\substack{j=1 \\ (j \neq i)}}^{N} a_{ij}\, x_j = b_i. \tag{4.45}$$

The solutions of (4.45) are therefore given by

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ (j \neq i)}}^{N} a_{ij}\, x_j \right). \tag{4.46}$$

4.5 Finding roots

The objective of this section is to introduce a numerical tool for finding root of a transcendental equation $f(x)$, that is, the $\tilde{x}$ value for which $f(\tilde{x}) = 0$. A brute force method is the so-called bisection method, which cuts the interval in half in which the root lies, as shown in figure 4.5. This procedure is repeated until either $f(x)$, evaluated at every new midpoint, is smaller than a prescribed tolerance.

A more elegant and efficient method to find the root of a function is Newton's method. Here one starts from a given initial guess value, x_0, for the root. A refined value for the guess value, x_1, is computed based on the x-intercept of the line tangent to f at x_0, as shown in figure 4.5. Mathematically this is expressed as

$$(f(x_1) - f(x_0)) = f'(x_0)\, (x_1 - x_0). \tag{4.47}$$

Since $f(x_1) = 0$ by construction, it follows from (4.47) that

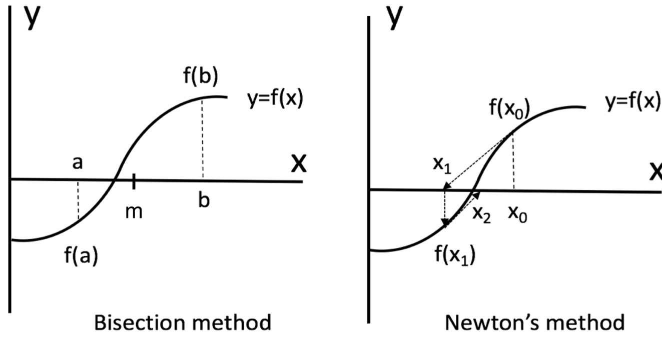


Figure 4.5. Left: The bisection method determines the midpoint of the interval in which the root of $f(x)$ lies. Right: Newton's method uses line tangents to f to find the root of $f(x)$.

$$0 - f(x_0) = f'(x_0)\,(x_1 - x_0) \quad \Rightarrow \quad x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}, \tag{4.48}$$

where x_1 is the new, improved value for root. This process is repeated according to the scheme ($i \in \mathbb{N}_+$)

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \tag{4.49}$$

until an x_{i+1} value is found for which $|f(x_{i+1})| < \epsilon$, where ϵ is a prescribed tolerance.

4.6 Solving ordinary differential equations

An ordinary differential equation (ODE) of order n is an equation of the form

$$F(x; y, y^{(1)}, y^{(2)}, \dots, y^{(n)}) = 0, \tag{4.50}$$

where y is a function of x, $y^{(1)} = dy/dx$, and $y^{(n)} = d^n y/dx^n$. In general, equation (4.50) has n linearly independent solutions, which are determined by initial conditions $Y_0 \equiv y(0)$, $Y_1 \equiv y^{(1)}(0)$, …, $Y_{n-1} \equiv y^{(n-1)}(0)$. In general, higher-order ODEs can be reduced to a set of first-order ODEs, which are easier to solve numerically. To demonstrate how this works, let us consider the second-order OED

$$P(x)\,\frac{d^2y(x)}{dx^2} + Q(x)\,\frac{dy(x)}{dx} = S(x), \tag{4.51}$$

where $P(x)$, $Q(x)$, and $S(x)$ are given functions of x. This equation has two linearly independent solutions, $y_1(x)$ and $y_2(x)$, with initial conditions Y_0 and Y_1. The functions $y_1(x)$ and $y_2(x)$ can be computed from the following set of coupled, first-order differential equations

$$\frac{dy_1(x)}{dx} = y_2(x), \tag{4.52}$$

$$\frac{dy_2(x)}{dx} = \frac{1}{P(x)}\big(S(x) - Q(x)y_2(x)\big), \tag{4.53}$$

which is equivalent to (4.50), as can bee seen by setting $y_1(x) = y(x)$ and $y_2(x) = dy_1/dx$. Similarly, a linear nth-order ordinary differential equation can be reduced to a system of n coupled first-order ODEs, according to the scheme

$$\frac{dy_k(x)}{dx} = f_k\big(y_1(x), y_2(x), \ldots, y_n(x)\big), \quad (k = 1, 2, \ldots, n), \tag{4.54}$$

with initial conditions Y_0, Y_1, …, Y_{n-1}. Common numerical methods for solving initial value problems of ordinary differential equations are the Euler method, the midpoint method, and the Runge–Kutta method. These methods will be introduced next.

4.6.1 The Euler method

The Euler method is the simplest method to approximate first-order derivatives. It advances the solution of a first-order ODE of the form $y' = f(x, y(x))$ (where $y' \equiv dy/dx$) with a given initial condition $Y_0 = y(0)$ from x to $x + h$ according to

$$y(x + h) = y(x) + h\, f(x, y(x)). \tag{4.55}$$

Here h is a properly chosen, small marching step which advances the solution form x to $x + h$. To indicate that the values of x are discretized numerically, it is convenient to write (4.54) as

$$y(x_{n+1}) = y(x_n) + h\, f(x_n, y(x_n)) \tag{4.56}$$

$$= y(x_n) + k_1, \tag{4.57}$$

where $k_1 \equiv h\, f(x_n, y(x_n))$. Euler's formula (4.57) can easily be solved numerically, but it is limited in practical usage since it uses the derivative information only at the beginning of an interval, which implies numerical errors that may grow quickly, depending on the behavior of $y(x)$.

To illustrate how Euler's method is used to solve a second-order differential equation, let us consider a small object of mass m which moves vertically (z-direction) through a viscous medium. The medium exerts a frictional force on the mass described by $b\, \dot{z}|\dot{z}|$, where b is a constant and the speed of the mass is given by $\dot{z} = dz/dt$. The gravitational force acting on the sphere is given by $-m\, g$, with g denoting the gravitational acceleration. The equation of motion of the mass follows from Newton's law and is given by

$$m\ddot{z} = -mg - b\dot{z}|\dot{z}| \tag{4.58}$$

Let us assume that at the initial time, $t = 0$, the mass is at $z(0) = 0$ and its initial velocity is $\dot{z}(0) = 10$ m s^{-1}, which corresponds to $Y_0 = 0$ and $Y_1 = 10$ m s^{-1}, respectively. The system of first-order ODEs associated with (4.58) is obtained as follows,

$$v = \dot{z} \Rightarrow z(t + \Delta t) = z(t) + v(t)\,\Delta t, \tag{4.59}$$

$$\dot{v} = \ddot{z} \Rightarrow v(t + \Delta t) = v(t) - \frac{1}{m}(m\,g + b\,v(t)\,|v(t)|)\Delta t, \tag{4.60}$$

where Δt is a properly chosen, small time step which advances the solution $z(t)$ from t to $t + \Delta t$. Since the equations are solved for discretized times and positions, it is appropriate to write (4.58) and (4.59) as

$$z_{i+1} = z_i + v_i\,\Delta t, \tag{4.61}$$

$$v_{i+1} = v_i - \frac{1}{m}(m\,g + b\,v_i\,|v_i|)\,\Delta t. \tag{4.62}$$

A numerical sample code which solves (4.61) and (4.62) subject to the boundary conditions $Y_0 = 0$ and $Y_1 = 10$ m s^{-1} is shown below.

```
z_1 = 0.0; v_1 = 10.0 !Define initial conditions
DO WHILE (z_2 >= 0.0)
     z_2 = z_1 + v_1*dt
     v_2 = v_1 – dt*(m*g + b*v_1*ABS(v_1)) / m
     time=time + dt
END DO
```

4.6.2 The midpoint method

The midpoint method, also known as the second-order Runge–Kutta method, improves the Euler method by adding a midpoint in the step, which increases the numerical accuracy. Equation (4.57) is then replaced by

$$y(x_{n+1}) = y(x_n) + k_2, \tag{4.63}$$

where k_1 and k_2 are given by

$$k_1 = h\,f(x_n, y_n), \tag{4.64}$$

$$k_2 = h\,f\left(x_n + \frac{h}{2}, y(x_n) + \frac{k_1}{2}\right). \tag{4.65}$$

4.6.3 The Runge–Kutta method

The Runge–Kutta method solves an ODE of the form $y' = f(x, y(x))$ by determining the value of $y(x + h)$ in terms of $y(x)$ computed at several different x values. To illustrate the method, let us begin with writing $y(x + h)$ as

$$y(x + h) = y(x) + (y(x + h) - y(x)) \tag{4.66}$$

$$= y(x) + \int_x^{x+h} y'(s)\, ds, \tag{4.67}$$

where $y' = dy/ds$. Defining $s = x + \tau\, h$ so that $ds = h\, d\tau$. This leads for the integral in (4.67) to

$$\int_x^{x+h} y'(s)\, ds = \int_0^1 y'(x + \tau h)\, d\tau \tag{4.68}$$

so that (4.67) can be written as

$$y(x + h) = y(x) + h \int_0^1 y'(x + \tau h)\, d\tau. \tag{4.69}$$

Next we approximate the integral in (4.68) by a finite sum,

$$\int_0^1 y'(x + \tau h)\, d\tau = \sum_{i=1}^{m} b_i\, y'(x + c_i h), \tag{4.70}$$

where the b_i denote unknown expansion coefficients. For $y' \equiv 1$ the values of these coefficients are constraint by the condition

$$\sum_{i=1}^{m} b_i = 1. \tag{4.71}$$

Substituting (4.70) into (4.69) leads to

$$y(x + h) = y(x) + h \sum_{i=1}^{m} b_i\, y'(x + c_i h) \tag{4.72}$$

$$= y(x) + h \sum_{i=1}^{m} b_i\, f(x + c_i h,\, y(x + c_i h)). \tag{4.73}$$

To evaluate (4.73) further, we need to find an approximate expression for y at the new grip points $x + c_i h$. With this in mind we make use of (4.69) to arrive for $y(x + c_i h)$ at

$$y(x + c_i h) = y(x) + h \int_0^{c_i} y'(x + \tau h)\, d\tau. \tag{4.74}$$

Approximating the integral in (4.74) by a finite sum, as in (4.70), leads to ($i = 1, \ldots, m,\ j = 1, \ldots, m$)

$$\int_0^{c_i} y'(x + \tau h)\, d\tau = \sum_{j=1}^{m} a_{i,j}\, y'(x + c_j h). \tag{4.75}$$

As before, for $y' \equiv 1$ we obtain the conditions

$$\sum_{j=1}^{m} a_{i,j} = c_i. \tag{4.76}$$

Substituting (4.75) into (4.74) leads to

$$y(x + c_i h) = y(x) + h \sum_{j=1}^{m} a_{i,j}\, y'\big(x + c_j h\big) \tag{4.77}$$

$$= y(x) + h \sum_{j=1}^{m} a_{i,j}\, f\big(x + c_j h,\, y(x + c_j h)\big). \tag{4.78}$$

Lastly, in order to simplify the notation, we introduce the abbreviation

$$\tilde{k}_j \equiv f\big(x + c_j h,\, y(x + c_j h)\big). \tag{4.79}$$

equation (4.78) can then be written in the more compact form

$$y(x + c_i h) = y(x) + h \sum_{j=1}^{m} a_{i,j}\, \tilde{k}_j, \quad (i = 1, \ldots, m). \tag{4.80}$$

Plugging (4.80) back into (4.79) leads to

$$\tilde{k}_j = f\left(x + c_j h,\, y(x) + h \sum_{l=1}^{m} a_{j,l}\, \tilde{k}_l\right), \quad (j = 1, \ldots, m). \tag{4.81}$$

The set of Runge–Kutta equations of order m follows from (4.73), which, by means of (4.79) and (4.81), can be written in the final form

$$y_{n+1} = y_n + h \sum_{i=1}^{m} b_i\, \tilde{k}_i, \tag{4.82}$$

$$\tilde{k}_i = f\left(x_n + c_i h,\, y_n + h \sum_{j=1}^{m} a_{i,j}\, \tilde{k}_j\right), \tag{4.83}$$

where $y_n \equiv y(x_n)$ and $y_{n+1} \equiv y(x_n + h)$. For $m = 4$, the unknown coefficients c_i, b_i, and $a_{i,j}$, summarized schematically as

$$\begin{array}{c|cccc} c_1 & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ c_2 & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ c_3 & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ c_4 & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \\ \hline & b_1 & b_2 & b_3 & b_4 \end{array} \tag{4.84}$$

have the following values,

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & 1/6 & 2/6 & 2/6 & 1/6 \end{array} \tag{4.85}$$

This leads for $\tilde{k}_1$, $\tilde{k}_2$, $\tilde{k}_3$, and $\tilde{k}_4$ to

$$
\begin{aligned}
\tilde{k}_1 &= f(x_n, y_n), \\
\tilde{k}_2 &= f\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}h\tilde{k}_1\right), \\
\tilde{k}_3 &= f\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}h\tilde{k}_2\right), \\
\tilde{k}_4 &= f\left(x_n + h, y_n + h\tilde{k}_3\right), \\
y_{n+1} &= y_n + \frac{1}{6}h(\tilde{k}_1 + 2\tilde{k}_2 + 2\tilde{k}_3 + \tilde{k}_4).
\end{aligned}
\tag{4.86}
$$

It is convenient to define $k_i \equiv h\tilde{k}_i$, the equations of the fourth-order Runge–Kutta method, which is by far the most common method to solve ODE, can be summarized as follows:

$$
\begin{aligned}
k_1 &= hf(x_n, y_n), \\
k_2 &= hf\left(x_n + \frac{h}{2}, y(x_n) + \frac{k_1}{2}\right), \\
k_3 &= hf\left(x_n + \frac{h}{2}, y(x_n) + \frac{k_2}{2}\right), \\
k_4 &= hf(x_n + h, y(x_n) + k_3), \\
y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4).
\end{aligned}
\tag{4.87}
$$

4.6.4 Boundary value problems

In the previous sections, we looked at ODEs whose solutions and derivatives have specific values at given points, such as position and velocity at an initial time. Such problems are referred to as initial value problems. This is different for so-called boundary value problems, where the solutions are required to have specific values at the boundaries of the system that is being studied.

As an example, let us consider a particle of mass m moving along the x-axis under the action of a time-dependent force $F(t)$. At time zero the particle is located at $x(0) = 0$. The the final time, t_f, the particle is at $x(t_f) = b$. The particle's motion is therefore described by the solution to the boundary value problem

$$
m\frac{d^2x}{dt^2} = F(t), \quad \text{where } x(0) = 0, \; x(t_f) = b. \tag{4.88}
$$

To determine the motion of the particle from $x(0) = 0$ to $x(t_f) = a$ numerically, we approximate the second-order time derivative in (4.88) by the finite-difference expression (4.23). This leads for (4.88) to

$$x_{i-1} - 2x_i + x_{i+1} = \tilde{f}_i, \quad x_0 = 0, \quad x_N = b, \tag{4.89}$$

where $\tilde{f}_i \equiv F_i \Delta t^2 / m$ and $F_i \equiv F(t_i)$. As can be seen from (4.89), the second-order differential equation is to be evaluated at three successive positions along the discretized x-axis, i.e. $x_{i-1} = x(t_{i-1})$, $x_i = x(t_i)$, and $x_{i+1} = x(t_{i+1})$. The discretized times are given by $t_i \equiv i\Delta t$ where $i = 1, 2, \ldots, N - 1$. The time segment Δt is defined as $\Delta t \equiv t_f / N$, with N denoting the total number of time segments of size Δt. Finally, the boundary values at the two end points require that $x_0 = 0$ and $x_N = b$, as shown in (4.89). Equation (4.89) constitutes a system of $N - 1$ coupled linear equations,

$$\begin{aligned}
i = 1: &\qquad x_0 - 2x_1 + x_2 = \tilde{f}_1, \\
i = 2: &\qquad x_1 - 2x_2 + x_3 = \tilde{f}_2, \\
i = 3: &\qquad x_2 - 2x_3 + x_4 = \tilde{f}_3, \\
\cdots &\qquad \cdots \\
i = N - 2: &\qquad x_{N-3} - 2x_{N-2} + x_{N-1} = \tilde{f}_{N-2}, \\
i = N - 1: &\qquad x_{N-2} - 2x_{N-1} + x_N = \tilde{f}_{N-1},
\end{aligned} \tag{4.90}$$

which determine the $N - 1$ unknown positions $x_1, x_2, \ldots, x_{N-1}$. Written in matrix form, this system of equations if given by

$$\begin{pmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{N-2} \\ x_{N-1} \end{pmatrix} = \begin{pmatrix} \tilde{f}_1 - x_0 \\ \tilde{f}_2 \\ \tilde{f}_3 \\ \vdots \\ \tilde{f}_{N-2} \\ \tilde{f}_{N-1} - x_N \end{pmatrix}, \tag{4.91}$$

where the coefficient matrix is tridiagonal and symmetric. Such systems can be solved numerically with standard numerical routines such as `tridag` from the *Numerical Recipes* [1, 2].

References

[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)

Omair Zubairi and Fridolin Weber

Chapter 5

Problem solving methodologies

Computational physics is a subject where computing is used to gain insight into complex systems. It is highly multidisciplinary involving physics, mathematics, and computer science, as illustrated in figure 5.1.

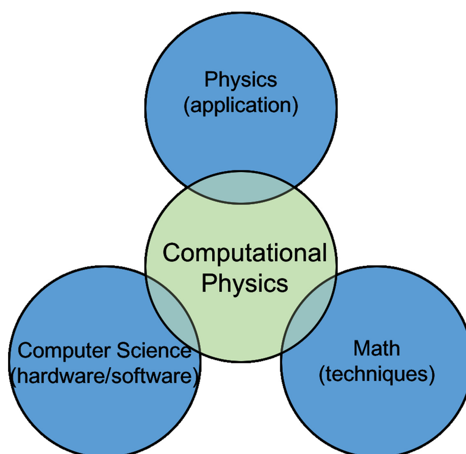


Figure 5.1. Venn diagram illustrating a multidisciplinary approach to computational physics.

This requires knowledge in the Linux/Unix environment, a programming language of some sort, and numerical techniques. Having this knowledge from the previous chapters, we can now attack a variety of problems in science and engineering which will require computing of some sort. In this chapter, we lay out a general guideline on methods and strategies which can be applied to any type of computational problem.

doi:10.1088/978-1-6817-4896-2ch5 © Morgan & Claypool Publishers 2018

When solving various problems in physics (and other scientific and engineering disciplines), we have to first ask ourselves a few questions:

- What is the physics problem at hand?
- What is the mathematical model which describes this particular problem?
  - What does the model tell us about the physics problem?
  - Is it realistic?
  - Can a solution be obtained in a reasonable amount of time and effort?
- Which programming language would work best for the application?

Asking the above questions is the first step to solving complex computational types of problems. Once you have determined the answers to these questions, you are ready to apply your skills in computing to solve the problem at hand.

5.1 General guidelines

A general guideline to solving computational physics problems is as follows:

1. Analyze the problem and think about it rationally.
2. Plan out the program, write out what you think you need to do.
3. Draw a flowchart explaining features of your program.
4. Write pseudo code to match your flow diagram.
5. Write your source code.
6. Compile your source code.
7. Execute (run) your source code (debug if necessary).

The seven steps listed here are very useful and can be applied not only to physics problems but many other problems in science and engineering. A great example of applying these guidelines is on the topic of two-dimensional kinematics, particularly on projectile motion. Projectile motion is a topic encountered by many students in a first semester physics course and can be applied to a variety of other applications.

5.2 Projectile motion example

Consider an object which is projected upward with some initial velocity v_0 at some given angle θ. For this scenario, we can ask the following questions:

- What is the max height the object reaches?
- How long will the object be in the air before it hits the ground?
- How far away is the object when it hits the ground?

To answer these questions, we can look at the trajectory of this object and computationally compute this by applying the seven steps listed above.

1. Analyze the problem and think about it rationally.

To solve this problem, we would need mathematical expressions which will describe the x and y positions. This would require us to use the well-known kinematic equations as described in the expressions in (5.1)–(5.3).

$$v_x = v_{x0} + a_x t \qquad v_y = v_{y0} + a_y t, \tag{5.1}$$

$$x = x_0 + v_{x0}t + \frac{1}{2}a_x t^2 \qquad y = y_0 + v_{y0}t + \frac{1}{2}a_y t^2, \tag{5.2}$$

$$v_x^2 = v_{x0}^2 + 2a_x(x - x_0) \qquad v_y^2 = v_{y0}^2 + 2a_y(y - y_0). \tag{5.3}$$

Since we are looking for the x and y positions, we will utilize the expressions

$$x = x_0 + v_{x0}t + \frac{1}{2}a_x t^2, \qquad y = y_0 + v_{y0}t + \frac{1}{2}a_y t^2, \tag{5.4}$$

where, if we assume if the initial positions in the x- and y-direction are zero, in addition to knowing the horizontal acceleration a_x is zero and the vertical acceleration a_y is simply the acceleration due to gravity, our expressions which describe the x and y positions modify to

$$x = v_0 \cos(\theta)t \tag{5.5}$$

$$y = v_0 \sin(\theta)t - \frac{1}{2}gt^2 \tag{5.6}$$

where we have substituted the expressions for v_{0x} and v_{0y} with $v_0 \cos(\theta)$ and $v_0 \sin(\theta)$, respectively. This is an important concept since velocity is a vector and must be broken up into its components.

2. Plan out the program.

Having our expressions which will describe the x and y positions, we can now start to plan out our program. As the projectile moves, the horizontal and vertical positions will change over time which will require the program to *update* these positions. Thus,

- updating the positions would require some sort of *iterative* process (i.e. loops).
- it would also need some sort of *step-size* (i.e. a time step).
- it would need to *write* these data out as they are being calculated.
- it would also need to update the step-size for each *new* position.
- lastly, we would need iterate this *until* the projectile hits the ground (i.e. requires a logical condition).

3. Draw out a flowchart.

A typical flow chart for this type of problem is illustrated in figure 5.2. The flow chart is describing the important points of the program, such as input parameters, iterative processes for our positions, writing data, and logical conditions.

4. Write out pseudo code.

We can easily write some simple pseudo code to describe the main processes happening within our program. The pseudo code below describes what will be happening within our iterative process.

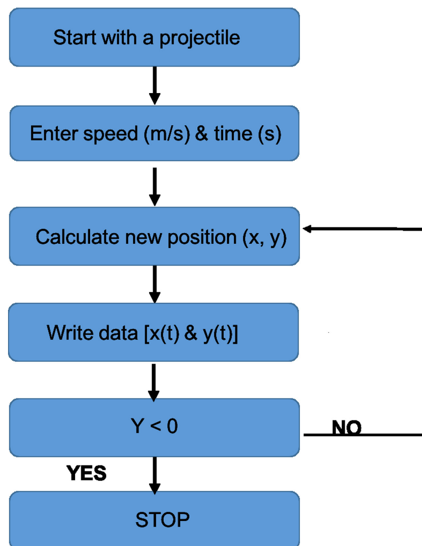


Figure 5.2. Flow chart illustrating calculations of projectile motion.

```
Do i = start, finish
     time =...
     x=...
     y=...
     Write out data
     Update time–step
     Iterate UNTIL Projectile Hits the Ground
End Do
```

5. Write your source code.

Using the pseudo code, one can easily see the major part of the program is within the loop. The actual code is left to the student to exercise their programming skills; however, the main loop is provided for your reference:

```
Do i = 0,1000
     t = i*dt

     x = v0*cos(theta)*t
     y = v0*sin(theta)*t – 0.5*g*t**2

     t = t + dt
     Write(15,*) x, y
     If (y <0) Exit
End Do
```

Once you have successfully written the entire source code, you can compile and run the program and graphically illustrate the trajectory of an object projected upward with some initial velocity and given angle.

Figure 5.3 shows the trajectories for a projectile launched with various velocities and angles by computing the expressions in (5.4) via the main loop listed above.

By applying our steps outlined in this chapter, we can attack a variety of physics problems. The worksheets and homework assignments listed in the next two chapters will give ample opportunities to apply these general guidelines.

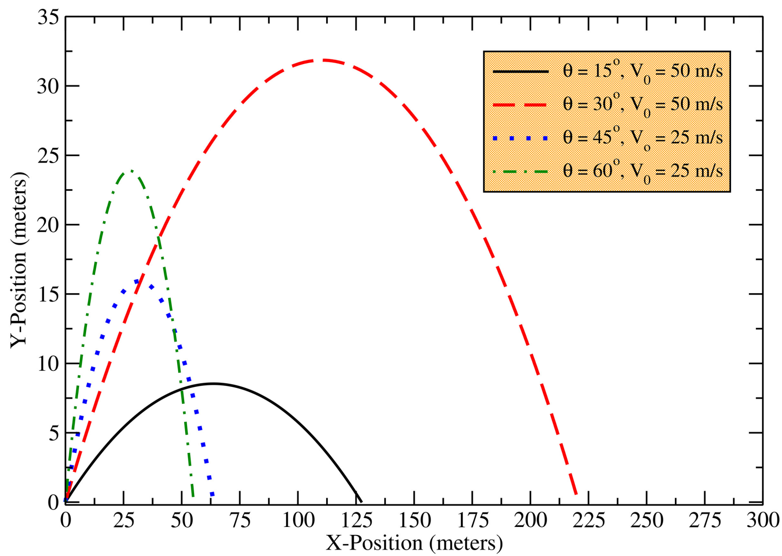


Figure 5.3. Horizontal (x) and (y) positions of an object projected upward with various initial velocities and angles.

Chapter 6

Worksheet assignments

6.1 Coding a mathematical expression

Purpose: Compute the value of a given mathematical expression.

Given is the following mathematical expression:

$$\chi = \left(\pi + \sin(x)\cos^3(x)\sqrt{1 + \sqrt{x}} + e^{-x\sin(x)}\right)(1 + x^2 + x^3 + x^4)^{-2} + \pi^2.$$

Tasks:

Write a structured Fortran 90 program which computes the function above for a given value of x.

Program design:

1. The variable x is keyboard input.
2. Make use of the Fortran 90 intrinsic functions `SIN(X)`, `COS(X)`, `SQRT(X)`, and `EXP(X)`.
3. Use the `Parameter` statement for π.
4. The value of χ is terminal output.
5. Run your code for $x = 0, 0.5, 1.0, 1.5$ and compare the results for χ with the results obtained by your fellow students.

6.2 Comparing two functions

Purpose: Practice the use of DO loops and the OPEN statement and generate graphical output.

Given are the following two functions,

$$\begin{aligned}\phi(x) &= e^{-x^2\sin(x)^2}x^{3/2},\\ \tau(x) &= 0.124523 + 0.739594(-0.25 + x) + 0.65781(-0.25 + x)^2\\ &\quad - 0.916955(-0.25 + x)^3\\ &\quad - 0.214698(-0.25 + x)^4 - 2.35154(-0.25 + x)^5,\end{aligned}$$

where $\tau(x)$ is the Taylor expansion of $\phi(x)$ for $x \in [0, 1]$.

doi:10.1088/978-1-6817-4896-2ch6

© Morgan & Claypool Publishers 2018

Tasks:
Write a structured Fortran 90 program which computes $\phi(x)$ and $\tau(x)$ for a range of x values.

Program design:

1. Use the range $x = 0,1$ with a step-size of 0.001.
2. Using the `OPEN` statement, design your code such that the results for $\phi(x)$ and $\tau(x)$ are written to two different output (data) files.
3. Declare the purpose of your code and all Fortran variables in the preamble of the program.
4. Produce a plot which shows $\phi(x)$ and $\tau(x)$ for $0 \leqslant x \leqslant 1$.

6.3 Bessel functions of the first kind

Purpose: Practice the DO loop and OPEN constructs and generate graphical output.

Bessel functions are used in optics to characterize the pattern you see when light is focused by a perfect lens with a circular aperture. The αth-order Bessel functions of the first kind, denoted as $J_\alpha(x)$, are solutions of Bessel's differential equation that are finite at the origin ($x = 0$) for integer or positive α, and diverge as x approaches zero for negative non-integer α values. It is possible to define the function by its Taylor series expansion around $x = 0$,

$$J_\alpha(x) = \sum_{p=0}^{\infty} \frac{(-1)^p}{p!\,\Gamma(p + \alpha + 1)}\left(\frac{x}{2}\right)^{2p+\alpha}, \tag{6.1}$$

where $\Gamma(n) = (n - 1)!$ $(n > 0)$ is the gamma function. The zeroth-order Bessel function, $J_0(x)$, follows from (6.1) as

$$J_0(x) \approx \sum_{p=0}^{m} \frac{(-1)^p}{p!\,\Gamma(p + 1)}\left(\frac{x}{2}\right)^{2p}. \tag{6.2}$$

Tasks:
Write a structured Fortran 90 program that takes the first 31 terms (i.e. $m = 30$) in the Taylor expansion (6.2) and a value for x (both keyboard input) to return (terminal output) a value for the zero-order Bessel function at that point.

You will need to increase the numerical precision when computing the factorials and carrying out the summation over p. This is accomplished by making changes in the `IMPLICIT NONE` construct, as shown below:

```
IMPLICIT NONE
INTEGER, PARAMETER :: DP = SELECTED_REAL_KIND(14)
REAL(KIND = DP)   :: ... list of your variables ...
.
.
. [your code]
.
.
```

Program design:

1. Making use of the `DO` loop construct, modify your program such that the value of $J_0(x)$ is computed over the range $0 \leqslant x \leqslant 20$ in steps of $\Delta x = 0.1$. Use the `OPEN` construct to write the results for $J_0(x)$ to an external file.
2. The more terms you use when calculating $J_0(x)$ the closer to the 'real' value your results should be. To demonstrate this, compute $J_0(x)$ $(0 \leqslant x \leqslant 20)$ for $m = 20, 22, 24, 26, 30$.
3. Illustrate the results of questions 1 and 2 graphically.
4. Extend your program such that the values of $J_\alpha(x)$ of (6.1) are computed, for a given value of α, over the range $0 \leqslant x \leqslant 20$ in steps of $\Delta x = 0.1$. Choose $m = 30$. The result is to be written to an external file. Run your code for $\alpha = 0, 1, 2, 3, 4, 5$ ($m = 30$ in each case) and compare the results graphically on a single plot.

6.4 Logical IF statements

Purpose: Practice logical IF statements and use Fortran intrinsic functions.

The following function $\Phi(x, y, z)$ is given for various conditions.

$$\Phi(x, y, z) = \begin{cases} \sqrt{x^3 + y^3 + z^3} & \text{if } x < 0 \\ \pi/4 & \text{if } x = 0, \\ \sin(xy) + \cos(xz) & \text{if } x > 0 \end{cases} \tag{6.3}$$

Tasks:

Write a structured Fortran 90 program that reads in three real numbers (x, y, z) and computes the following function $\phi(x, y, z)$ for the three conditions listed above.

Program design:

1. Comment the different steps in your program.
2. Make use of the logical `IF` statement to discriminate between $x < 0$, $x = 0$, and $x > 0$.
3. Make use of the `parameter` statement for π.
4. Run your the program for the following values:

$$\begin{aligned}(x, y, z) &= (-1.5, 4.0, 9.0)\\ &= (0.0, 12.0, -2.2)\\ &= (3.5, 4.0, 0.5)\end{aligned}$$

6.5 Lead concentration in humans (data analytics)

Purpose: Practice DO loops, logical IF statements, and Input/Output data handling.

Lead is widely present in our environment due to its natural occurrence and human activities that have introduced it into the general environment. Because lead may be present in environments where food crops are grown and animals used for food are raised, various foods may contain unavoidable but small amounts of lead

that do not pose a significant risk to human health. Small amounts of lead in adults are not thought to be harmful. However, even low levels of lead can be very dangerous to infants and children. According to the Centers for Disease Control and Prevention (CDC), blood lead levels of 2.4 μmol/10 L (5 μg dL^{-1}) or greater require further testing and monitoring in children[1].

For this worksheet, you will write a Fortran program which reads lead concentration data measured in children from a data file. For this data set, the program will then calculate:

the mean $\langle x \rangle$,

$$\langle x \rangle = \frac{1}{n} \sum_{i=1}^{n} x_i, \tag{6.4}$$

the variance σ^2,

$$\sigma^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \langle x \rangle)^2, \tag{6.5}$$

and the standard deviation σ,

$$\sigma = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \langle x \rangle)^2}, \tag{6.6}$$

where n is the number of measured data (observations). Recall that σ is a very useful measure of the scatter of observed data, since a range covered by 1σ above $\langle x \rangle$ and one σ below the mean includes about 68% of the observations, a range of 2 σ above and two below $\langle x \rangle$ is about 95% of the observations, and a range of 3σ above and three below $\langle x \rangle$ is about 99.7% of the observations. Consequently, by putting one, two, or three standard deviations above and below the mean one can estimate the ranges that would be expected to include about 68%, 95%, and 99.7% of the observed data.

The probability distribution (function)

$$f(x \mid \langle x \rangle, \sigma^2) = \frac{1}{2\sigma^2 \pi} e^{-(x-<x>)^2/2\sigma^2} \tag{6.7}$$

of a set of data is represented by a curve defined uniquely by two parameters, which are the mean and the standard deviation of a given data set. The curve is always symmetrically bell shaped, but the extent to which the bell is compressed or flattened out depends on the standard deviation of a given data set.

Tasks:

Write a structured Fortran program which reads lead concentrations (in μmol/10 L) measured in children from an external data file and calculates the mean value, the

[1] 5 μg dL^{-1} stands for 5 micrograms per deciliter (dL).

variance, and the standard deviation, as defined in (6.4)–(6.6). The data for this set are given by the following array:

$$data = [0.1, 0.4, 0.6, 0.8, 1.1, 1.2, 1.3, 1.5, 1.7, 1.9, 1.9, 2.0, 2.2, 2.6, 3.2]$$

You will have to manually create an external file (`.dat`) for the data listed above which then you will `READ` into your program.

Program design:

1. The screen output generated by the program should be as follows:

```
The mean of this data set is:
The variance of this data set is:
The standard deviation is:
The number of data points is:
```

2. Make use the implicit `DO`-loop construct (that is, `READ(unitnumber, *, end=unitnumber)`) to read the data from the data file.
3. If the number of input data is less than 2, the program should tell (screen output) the user that the number of input data is insufficient to carry out a statistical analysis. Use the `CALL EXIT` statement to terminate the program.
4. The probability function $f(x|\langle x\rangle, \sigma^2)$ is to be computed and graphically illustrated for $-2\ \mu\text{mol}/10\ \text{L} \leqslant x \leqslant 5\ \mu\text{mol}/10\ \text{L}$. To cover this range, make use of $x_k = a + (b - a)k/N$, with $N = 100$.
5. In your plot, mark the locations where 68% ($\langle x\rangle \pm \sigma$), 95% ($\langle x\rangle \pm 2\sigma$), and 99.7% $\langle x\rangle \pm 3\sigma$ of the data are located.

6.6 Nested DO loops and double summations

Purpose: Practice the use of nested DO loops.

Tasks:

Write a structured Fortran 90 program which computes and prints out (standard output) the results of

$$A(N) \equiv \sum_{i=1}^{N}\sum_{j=1}^{i} j^{-2}(i+1)^{-2}, \tag{6.8}$$

$$B(N) \equiv \prod_{k=1}^{N-1} \sin(k\pi/N), \tag{6.9}$$

$$C(s; N) \equiv \pi\ s \prod_{k=1}^{N} (1 - s^2/k^2), \quad (s = 1.5). \tag{6.10}$$

for $N = 10{,}100$ with a step-size of 10. Compare your results for $A(N)$, $B(N)$, and $C(s;\, N)$ with the analytic results given by

$$A(\infty) = \sum_{i=1}^{\infty} \sum_{j=1}^{i} j^{-2}(i+1)^{-2} = \pi^4/120 = 0.8117, \tag{6.11}$$

$$B(N) = \prod_{k=1}^{N-1} \sin(k\pi/N) = 2^{1-N}\, N, \tag{6.12}$$

$$C(s;\, N) = \pi\, s \prod_{n=1}^{N} (1 - s^2/n^2) = \sin(\pi\, s). \tag{6.13}$$

Program design:

1. Make use of the `Parameter` statement for π.
2. The value of s is keyboard input.
3. Make use of the nested `DO` loop construct.
4. The (unformatted) terminal output produced by your code should be something like this:

```
N= 10
A=...       A(analytic)=...
B=...       B(analytic)=...
C=...       C(analytic)=...    (s=....)

N= 20
A=...       A(analytic)=...
B=...       B(analytic)=...
C=...       C(analytic)=...    (s=....)
.
.
.
N= 100
A=...       A(analytic)=...
B=...       B(analytic)=...
C=...       C(analytic)=...    (s=....)
```

6.7 Ionic crystals

Purpose: Practice the use of DO loops, IF statements, and formatted I/O.

Consider a collection of N charges brought together so that the ith particle of charge q_i is located at the position vector $\vec{r}_i$. Then, the potential energy of the assemblage is given by

$$U = \frac{1}{2}\sum_{i=1}^{N} q_i \, \Phi_i, \qquad \Phi_i = k_e \sum_{j=1}^{N} \frac{q_j}{|\vec{r}_i - \vec{r}_j|}, \tag{6.14}$$

where Φ_i is the electrostatic potential of all the charges (except the ith charge) at the location of the ith charge and k_e is a constant ($=1/4\pi\epsilon_0$). The potential energy (or binding energy, as it is usually called) per ion is given by

$$u = -\alpha \frac{k_e \, q^2}{a}, \tag{6.15}$$

where α denotes the Madelung constant. For a three-dimensional ionic crystal, α is given by

$$\begin{aligned} \alpha = & -6\sum_{j=1}^{n}\frac{(-1)^j}{j} - 12\sum_{j=1}^{n}\sum_{k=1}^{n}\frac{(-1)^{j+k}}{\sqrt{j^2+k^2}} \\ & -8\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\frac{(-1)^{i+j+k}}{\sqrt{i^2+j^2+k^2}}, \end{aligned} \tag{6.16}$$

where n is a large number (equal to $N^{1/3}/2$).

Tasks:
Write a structured Fortran 90 program which computes the Madelung number defined in (6.16) for n = 10, 20, 30, 40, 50, 100.

Program design:

1. Thoroughly comment on the iterative process within your program.
2. Use a `DO` loop to determine the value of n.
3. The results are to be written in tabulated and formatted form to standard output. The terminal output must be *exactly* as shown below:

```
Approximations for the Madelung constant in three dimensions:
           N   Madelung number alpha
         ____________________________
           10    Mad3D=   x.xxxxxxxxx
           20    Mad3D=   x.xxxxxxxxx

           . . .

           50    Mad3D=   x.xxxxxxxxx
           100    Mad3D=   x.xxxxxxxxx
```

6.8 Least-squares fit

Purpose: Fit a set of experimental data with a mathematical model (using the least-squares technique).

A satellite experiment was launched in the NIMBUS 7 spacecraft in 1978 to collect data on the composition and structure of the middle atmosphere [1, 2]. The instrumentation and sensors collected data from October 25, 1978, to May 28, 1979, returning more than 7000 sets of data to the Earth each day. These data were used to determine temperature, ozone, water, vapor, nitric acid, and nitrogen dioxide distributions in the stratosphere and mesosphere. (The stratosphere and the mesosphere are atmospheric layers around the Earth from about 15 km to approximately 85 km above the Earth's surface.) Assume that we have collected a set of data measuring the ozone mixing ratios in parts per million volume (ppmv) as shown in table 6.1.

Over small regions, these data are nearly linear, and thus we can use a linear model to estimate the ozone at altitudes other than the ones for which we have specific data.

Tasks:

Write a structured Fortran 90 program that reads a data file created by the data set from table 6.1 and performs a linear fit on the data as described in chapter 5 via the equations listed under section 4.1.1.

You will use the least-squares technique to determine and print the best-fit (linear) model data as well as the averaged squared error.

Program design:

1. The original data and the best-fit model data are to be written to an output file.
2. The screen output produced by your program should look as follows:

Table 6.1. Atmospheric data measuring ozone mixing ratios.

Altitude (km)	Ozone mixing ratios (ppmv)
20	3
22	4
24	4
26	5
28	6
31	8
33	7
34	9
37	8

```
Linear model:    y= . . .  x + . . .
original  original  estimated  residual
 x         y             y
20.00   3.00          . . .      . . .
22.00   4.00          . . .      . . .
24.00   4.00          . . .      . . .
26.00   5.00          . . .      . . .
28.00   6.00          . . .      . . .
31.00   8.00          . . .      . . .
33.00   7.00          . . .      . . .
34.00   9.00          . . .      . . .
37.00   8.00          . . .      . . .
Averaged squared error = . . .
```

3. Generate a plot which has the original data and the best-fit (least-squares) linear model in the same graph.

6.9 Numerical derivatives

Purpose: Illustrate how to numerically approximate derivatives.

The first derivative of a function can be approximated via the forward Euler's method and the three-point rule formula as described in chapter 4. While, these methods serve their purpose for approximating the first derivative, higher-order methods are required for higher-order derivatives of functions.

For second-order derivatives, one can use the three-point rule given by (4.23) and the five-point rule given by (4.24).

Tasks:

Write a well-commented structured FORTRAN 90 program which numerically approximates the second derivative for the function:

$$f(x) = x\sin(x) \tag{6.17}$$

at the point of $x_0 = 26.0$.

Program design:

1. Declare all variables with quad precision, i.e. `Real (kind=16)::`.
2. Choose a step-size such that `dx=0.1`.
3. Loop over five increments (i.e. `i=1, 5`).
4. Calculate the error between the exact solution and the approximations.
5. Produce a plot on a **log–log scale** which shows the results for the error versus step-size for both approximations.
6. Comment on your graph, what can you say about the slope for each?

Note: When you change your scale, you will have to re-scale your axis—be mindful of your data.

6.10 Numerical integration

Purpose: You will learn how to integrate continuous functions (such as polynomials, exponentials, and trigonometric functions) numerically using two so-called closed techniques for numerical integration—the trapezoidal rule and Simpson's rule.

Tasks:
Write a structured and well-commented Fortran 90 program that uses both the trapezoidal (4.32) and Simpson's (4.37) rules of numerical integration to compute the integral

$$w = \int_a^b (3e^{-x}\sin^2 x + 1)dx. \tag{6.18}$$

You will run your program for the following cases:
$N = 100$, $a = 0$, $b = 1$
$N = 100$, $a = 0$, $b = 2$
$N = 100$, $a = 0$, $b = 3$ (The exact result for this case is 4.14801).
$N = 100$, $a = 0$, $b = 4$
$N = 100$, $a = 0$, $b = 5$
$N = 100$, $a = 0$, $b = 10$
$N = 100$, $a = 0$, $b = 50$
$N = 100$, $a = 0$, $b = 100$

and explore the dependence of the numerical result(s) on N.

Program design:
1. Design your program such that it lets the user enter the integration boundaries (a and b) and the number of intervals (N).
2. The screen output generated by your program should display:

– the integration limits, a and b;
– the number of intervals, N;
– the result of the integral using the trapezoidal rule;
– the result of the integral using Simpson's rule.

6.11 Finding roots of a nonlinear equation

Purpose: You will learn how to compute the roots of a nonlinear equation numerically using Newton's method.

Given is the following nonlinear equation,

$$f(x) = e^x \ln(x) - x^2. \tag{6.19}$$

Tasks:
1. Write a Fortran 90 program which uses Newton's method as described in section 4.5 to compute the root(s) of (6.19).

Program design:

1. Design the program such that $f(x)$ and $f'(x)$ are computed in different `FUNCTION` subprograms.
2. Stop Newton's iteration scheme if $|x_{i+1} - x_i| \leqslant \epsilon$, where $\epsilon = 10^{-6}$.
3. Make sure that a message is written to standard output if the program fails to detect the root(s) of the equation.
4. Generate a plot $f(x)$ and check whether or not your solution(s) for $f(x) = 0$ is (are) correct.

6.12 Ordinary differential equations

Purpose: Illustrate how to solve an ordinary first-order differential equation numerically.

Given is the ordinary first-order differential equation

$$\cos(x)\, y'(x) + \sin(x) y(x) = 2\cos^3(x)\sin(x) - 1, \tag{6.20}$$

where $0 \leqslant x \leqslant 20$. The initial condition is $y(0) = 6.75$. The analytic solution of (6.20) is given by

$$y_{\text{analytic}}(x) = -0.2\cos(x)\cos(2x) - \sin(x) + 7\cos(x). \tag{6.21}$$

Tasks:

Write a structured Fortran 90 program which solves (6.20) numerically via Euler's method, as described in section 4.6.1. The outcome is to be compared graphically with the analytic result obtained from (6.21).

Program design:

1. Use a logical `IF` statement to assign a numerical value to $x \in [0, 20]$. The value to be used for the step size Δx is keyboard input.
2. Use the `OPEN` statement to write the results of (6.20) and (6.21) for $0 \leqslant x \leqslant 20$ to external output files.
3. Produce a plot which shows $y(x)$.
4. In the same plot, show $y_{\text{analytic}}(x)$ for $\Delta = 0.01$ and $\Delta = 0.001$.

6.13 Projectile in a viscous medium

Purpose: Illustrate how to solve ordinary differential equations numerically.

A spherical projectile moves vertically in a viscous medium near the surface of the Earth. It experiences two forces, the gravitational attraction of the Earth ($-m\, g$) and the viscous force F_v from the medium in which it moves. The former is directed downward (negative z) direction and the latter is directed opposite to the velocity v. Usually, the magnitude of the viscous force is a function of the speed with which the

projectile moves, symbolically, $F_v = f(v)$. Assuming that $f = b\,v^2$ ($b > 0$ denotes a positive constant), the projectile's equation of motion is then given by

$$m\,\frac{d^2z}{dt^2} = -m\,g - b\,\frac{dz}{dt}\left|\frac{dz}{dt}\right|. \tag{6.22}$$

The parameters in this equation are the projectile's mass, $m = 50$ g, and the gravitational acceleration, $g = 9.81$ m s^{-2}. The constant b is given by

$$b = \frac{1}{2}\,\rho\,C_d\,A, \tag{6.23}$$

where ρ is the density of the viscous medium, C_d is the drag coefficient,

$$C_d \approx \frac{24}{Re} + \frac{6}{1 + \sqrt{Re}} + 0.4, \tag{6.24}$$

A is the cross-sectional area, and Re denotes the Reynolds number[2].

Tasks:
Write a structured Fortran 90 program which solves the nonlinear second-order differential equation (6.22) using the midpoint method as described in section 4.6.2 for given values of m, g, $\rho = 1500$ kgm^{-3}, and $Re = 1.5 \times 10^4$. The radius of the projectile is $r = 5$ mm. The initial conditions of the projectile are $z_0 \equiv z(t = 0) = 0$ and $v_0 \equiv \dot{z}(t = 0) = 5$ m s^{-1}.

Program design:

1. The formatted screen output should be as follows:

```
Projectile's initial position (in meters):
Projectile's initial speed (in meters/seconds):

Stepsize delta t (in seconds): 0.0001
t=0.00 seconds, z(t)=0.00 meters
t=0.00 seconds, v(t)=5.00 meters/seconds
```

2. Assume a temporal step size of $\Delta t = 0.0001$ s.
3. Use `FUNCTION` subprograms to compute C_d, the area (cross section) of the spherical projectile, and the constant b.
4. Generate a plot which shows the projectile's position $z(t)$ and speed $v(t)$ as a function of time, t.

[2] The Reynolds number is a criterion of whether fluid, liquid or gas flow is steady (laminar) or unsteady (turbulent). If the Reynolds number is less than around 2000, flow is generally laminar. For Reynolds numbers greater than around 2000, flow is usually turbulent.

Explore the dependence of your numerical solutions on the value chosen for Δt by running your code for time steps $\Delta t = 0.1$, 0.01, and 0.001. Show the results in the plots that you just generated.

Additional tasks:
Modify your program to include the Runge–Kutta method as described in section 4.6.3 via the expressions in (4.87). Show the results for $z(t)$ and $v(t)$ graphically for the midpoint method and the Runge–Kutta method in one plot and comment on any differences.

6.14 Damped harmonic oscillator

Purpose: Illustrate how to solve higher-order differential equations numerically.

Given is an object of mass m attached to a spring (spring constant κ). The object oscillates back and forth in the x-direction, as shown graphically in figure 6.1.

The motion of m is damped by a frictional, velocity-dependent force $-\beta\dot{x}$, where β is a constant. The equation of motion of m is thus given by

$$m\ \ddot{x} + \kappa\ x + \beta\ \dot{x} = 0, \tag{6.25}$$

with $m = 2 \times 10^4$ g and $\beta = 14$ kg s^{-1}.

Tasks:
Write a structured and well-commented Fortran 90 program that solves equation (6.25) numerically via one or more of the methods described in sections 4.6.1–4.6.3. You will do this for both for $\beta = 0$ (i.e. no damping) and $\beta \neq 0$ and illustrate the results graphically.

From your results, determine an approximate value for the amplitude, A, and the period of oscillation, T.

The initial conditions are $x(0) = 4$ m and $\dot{x}(0) = -150$ cm s^{-1}. The acceleration of m at $t = 0$ is -10^3 cm s^{-2}.

Program design:

1. Design your code such that the user is prompted to key in a value for the time step Δt. The numerical results for $x(t)$ and $v(t)$ are to be written to output files.
2. Generate a plot for $x(t)$ and $v(t)$ for both the undamped as well as the damped case for $0 \leqslant t \leqslant 15$ s.
3. From your program, also numerically, determine the force, F_3, that acts on m at $t = 0.3$ s (i.e. determine $F_3(0.3)$).

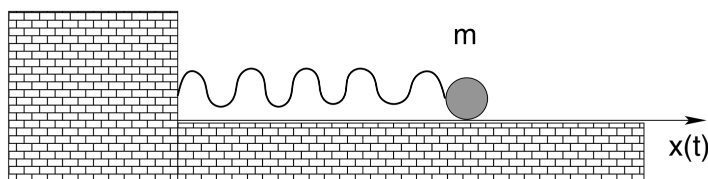


Figure 6.1. One-dimensional oscillator.

6.15 RLC circuit

Purpose: Illustrate how to solve higher-order differential equations numerically and learn the 90 module feature.

An RLC circuit [3] is an oscillating circuit consisting of a resistor (R), capacitor (C), and inductor (L) connected in series (see figure 6.2). The capacitor is charged initially. The voltage of this charged capacitor causes a current ($I = dq/dt$) to flow in the inductor to discharge the capacitor.

Once the capacitor is discharged, the inductor resists any change in the current flow, causing the capacitor to be charged again with the opposite polarity. The voltage in the capacitor eventually causes the current flow to stop and then flow in the opposite direction. The result is an oscillating electric current. The differential equation which describes the flow of the electric current throughout the RLC circuit is given by

$$\frac{d^2q}{dt^2} = -\frac{R}{L}\frac{dq}{dt} - \frac{q}{L\,C}, \tag{6.26}$$

where $L = 0.012$ H and $C = 1.0 \times 10^{-5}$ F[3].

Tasks:
Write (6.26) as a system of coupled first-order differential equations.

Write a structured Fortran 90 program which solves this system of equations numerically via one or more of the methods described in section 4.6.1–4.6.3 for times $0 \leqslant t \leqslant t_{\text{final}}$, where $t_{\text{final}} = 5 \times 10^{-3}$ s. The initial conditions for the electric charge and electric current are $q(t = 0) = 1.6 \times 10^{-5}$ C and $dq/dt = I(t = 0) = 0.01$ A, respectively.

Program design:
1. Choose $\Delta t = t_{\text{final}}/500$ for the temporal step size.
2. Use a `FUNCTION` to evaluate the right-hand side of (6.26).
3. Use the Fortran 90 `module` feature to assign values to L, C, $I(0)$, and $q(0)$.
4. Design your code such that the user is prompted to input the numerical value for R from keyboard.
5. Generate a plot which shows q as a function of t for $R = 1.5\ \Omega$, $R = 5\ \Omega$, and $R = 50\ \Omega$ (single plot).

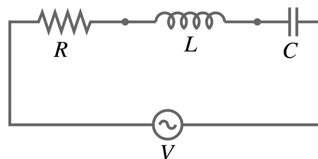


Figure 6.2. Illustration of an RLC circuit.

[3] $\text{H} = \Omega\,\text{s}$, $\text{F} = \text{s}\,\Omega^{-1}$.

References

[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)

Introduction to Computational Physics for Undergraduates

Omair Zubairi and Fridolin Weber

Chapter 7

Homework assignments

7.1 Fresnel coefficients

To study the reflection and transmission of light at a material interface (e.g. air–glass), one typically examines three distinct waves traveling in the directions depicted in figure 7.1, where $E^{(p)}$ and $E^{(s)}$ denote the components of the electric field in the parallel (p) and vertical (s) directions. The subscripts r, i, and t stand for reflected, incident, and transmitted, respectively. The index of refraction n_i characterizes the material on the left, and n_t characterizes the material on the right of the vertical axis. The incident plane wave makes an angle θ_i with the normal to the interface, the reflected wave makes an angle θ_r with the interface normal, and the transmitted plane wave makes an angle θ_t with the interface normal. The ratio of the reflected and transmitted electric field components to the incident field components are specified by the following coefficients, called Fresnel coefficients:

$$r_s \equiv \frac{E_r^{(s)}}{E_i^{(s)}} = \frac{n_i \cos\theta_i - n_t \cos\theta_t}{n_i \cos\theta_i + n_t \cos\theta_t}, \tag{7.1}$$

$$t_s \equiv \frac{E_t^{(s)}}{E_i^{(s)}} = \frac{2n_i \cos\theta_i}{n_i \cos\theta_i + n_t \cos\theta_t}, \tag{7.2}$$

$$r_p \equiv \frac{E_r^{(p)}}{E_i^{(p)}} = \frac{n_i \cos\theta_t - n_t \cos\theta_i}{n_i \cos\theta_t + n_t \cos\theta_i}, \tag{7.3}$$

$$t_p \equiv \frac{E_t^{(p)}}{E_i^{(p)}} = \frac{2n_i \cos\theta_i}{n_i \cos\theta_t + n_t \cos\theta_i}. \tag{7.4}$$

The Fresnel coefficients allow one to easily connect the electric field amplitudes on the two sides at the material interface. They also keep track of phase shifts at a boundary.

doi:10.1088/978-1-6817-4896-2ch7 © Morgan & Claypool Publishers 2018

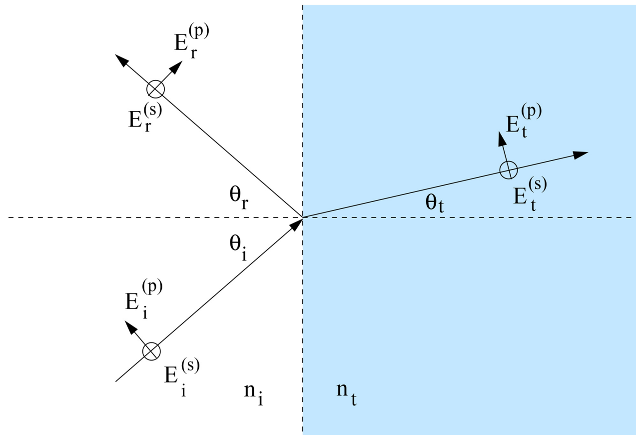


Figure 7.1. Incident (i), reflected (r), and transmitted (t) plane waves at a material (e.g. air–glass) interface.

Tasks:
Write a structured Fortran 90 program which computes (and outputs) the Fresnel coefficients given in (7.1) to (7.4) for $0 \leqslant \theta_i \leqslant 90°$.

Program design:

1. Use $n_i = 1$ and $n_t = 1.5$ (air–glass interface), and a fixed angle of $\theta_t = 5°$ for the transmitted light ray.
2. The program should also compute (and output) the reflectances $R_s \equiv r_s^2$ and $R_p \equiv r_p^2$, and the transmittances $T_s \equiv 1 - R_s$ and $T_p \equiv 1 - R_p$, for $0 \leqslant \theta_i \leqslant 90°$.
3. Generate a plot which shows the Fresnel coefficients r_s, t_s, r_p, and t_p for $0 \leqslant \theta_i \leqslant 90°$.
4. Generate a plot which shows R_s, R_p, T_s, and T_p for $0 \leqslant \theta_i \leqslant 90°$.

7.2 Earth atmosphere model

To help aircraft designers, standard atmosphere models of the variation of properties through the atmosphere have been developed. One such model is discussed here. The model assumes that the pressure and temperature change only with altitude. The model has three zones with separate curve fits for the troposphere, the lower stratosphere, and the upper stratosphere. The troposphere runs from the surface of the Earth to 11 km. In the troposphere, the temperature T decreases linearly and the pressure p decreases exponentially. The curve fits for the *troposphere* are given by

$$T = 15.04 - 0.006\,49\, h,$$
$$p = 101.29[(T + 273.1)/288.08]^{5.256},$$

where the temperature is given in Celsius degrees, the pressure in kilo-pascals (kPa), and h is the altitude in meters.

The lower stratosphere runs from 11 km to 25 km. In the lower stratosphere the temperature is constant and the pressure decreases exponentially. The curve fits for the *lower stratosphere* are

$$T = -56.46,$$
$$p = 22.65 \exp(1.73 - 0.000\,157\,h).$$

The upper stratosphere model is used for altitudes from 25 km to 50 km. In the upper stratosphere the temperature increases slightly and the pressure decreases exponentially. The curve fits for the *upper stratosphere* are

$$T = -131.21 + 0.002\,99\,h,$$
$$p = 2.488[(T + 273.1)/216.6]^{-11.388}.$$

In each zone the density ρ in kg m^{-3} is derived from the equation of state

$$\rho = p/[0.2869(T + 273.1)].$$

Tasks:
Write a structured Fortran 90 program that computes the atmosphere's temperature T (in °C), pressure p (in kPA), and density ρ (in kg m^{-3}) as a function of altitude h (in m). The altitude is from 0 to 50 000 m.

The results are to be illustrated graphically.

Program design:

1. Use `DO` loops to compute $T(h)$, $p(h)$, and $\rho(h)$ for each atmospheric layer (preferably use one `DO` loop).
2. Make use of logical `If–Then–Else If` conditions for the different regions of the atmosphere.
3. Use a vertical step size of $\Delta h = 100$ m (you will have to think about how much you will need to iterate to reach 50 000 m).
4. The results for the $T(h)$, $p(h)$, and $\rho(h)$ are to be written to three separate output files.
5. Generate plots which show the profiles of temperature, pressure, and density graphically.

7.3 Magnetic permeability

The magnetic field around a wire (figure 7.2) carrying an electric current I is given by

$$B(r) = \mu_0 I/(2\pi r),$$

where r is the distance measured from the wire and μ_0 is the magnetic permeability.

Experimental results of the measured magnetic field, B, as a function of r is given in table 7.1. The current is kept constant at 2.7 A.

Tasks:
Write a structured Fortran 90 program which determines the value of μ_0 (note that the units of the permeability are μT m A^{-1}.) using the *linear* (i.e. $f(x) = a + bx$) least-squares method, as described in section 4.1.1.

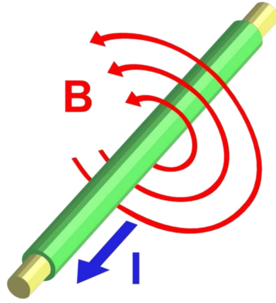


Figure 7.2. Magnetic field around a wire.

Table 7.1. Experimental measurement of magnetic field B at a certain distance from a current carrying wire.

Data points	1	2	3	4	5
r (cm)	10.0	20.0	30.0	40.0	50.0
B(μT)	5.4	2.7	1.8	1.4	1.0

Program design:

1. The `DATA` statement is used to input the values from table 7.1 into your code.
2. The coefficients a and b are computed in a subroutine named `SUMMATIONS`.
3. The permeability is computed by a function named `PERMEABILITY`.
4. The value of the permeability is written to standard output in the main program.
5. Generate a plot which shows the experimental data of table 7.1 as well as the least-squares linear fit result of the data.

7.4 Maxwell–Boltzmann distribution

The velocity distribution of gas molecules at temperature T is given by the Maxwell–Boltzmann distribution function,

$$P(v)dv = 4\pi\left(\frac{m}{2\pi kT}\right)^{3/2} \mathrm{e}^{-mv^2/2kT} v^2 \, dv. \tag{7.5}$$

From (7.5) one obtains

$$f(v,\, T) \equiv \int_0^v P(v') \, dv' = 4\pi\left(\frac{m}{2\pi kT}\right)^{3/2} \int_0^v \mathrm{e}^{-mv'^2/2kT} v'^2 \, dv' \tag{7.6}$$

for the fraction, f, of molecules having speed less than v. For a given speed and temperature, f has a value between 0 and 1. The quantities in (7.6) are as follows: m is the mass of the gas molecules in kg mol^{-1}, k is the Boltzmann constant, T is the temperature in kelvins, and v is the speed in m s^{-1}.

The Fortran 90 code provided in appendix C computes $f(v, T)$ for air molecule with speeds ranging from $v = 100\ \mathrm{m\,s^{-1}}$, $5000\ \mathrm{m\ s^{-1}}$ with a step-size of $(100\ \mathrm{m\ s^{-1}})$ for a given temperature T. Since air consists of $\chi_{\mathrm{N}} = 78\%$ nitrogen, $\chi_{\mathrm{O}} = 21\%$ oxygen, and $\chi_{\mathrm{Ar}} = 1\%$ argon, an average value of $m = \sum_{i=\mathrm{N,O,Ar}} m_i\, \chi_i$ is used or the mass of the gas molecules of air. The respective mole masses are $m_{\mathrm{N}} = 28\ \mathrm{g\ mol^{-1}}$, $m_{\mathrm{O}} = 32\ \mathrm{g\ mol^{-1}}$, and $m_{\mathrm{Ar}} = 40\ \mathrm{g\ mol^{-1}}$. Using certain defined quantities, one can easily simplify (7.6) by applying a change of variables.

Tasks:
Using the quantities $\beta = \sqrt{2kT/m}$ and $y^2 = v^2/\beta^2$ apply a change of variables and show that (7.6) can be written as

$$f(v,\ T) = \frac{4\pi}{(\pi)^{3/2}} \int_0^{y\beta} y^2 \mathrm{e}^{-y^2} \mathrm{d}y. \tag{7.7}$$

Modify the code given in appendix C (which currently uses the trapezoidal method) to include the Simpson's method of integration as described in section 4.3.2 to compute the integral given in (7.7).

Program design:

1. The Simpson's method and the trapezoidal method are to be written in separate subroutines.
2. The program will ask the user which method to use (i.e. user input for the integration methods).
3. Design your code such that the results for $f(v, T)$ for $0 \leqslant v \leqslant 5000\ \mathrm{m\ s^{-1}}$ are written to an output file for a fixed temperature. (Use $\Delta v = 10\ \mathrm{m\ s^{-1}}$.)
4. Generate a plot for $f(v, T)$ for temperatures: $T = 40\ °\mathrm{C}$, $2000\ °\mathrm{C}$, and $T = 5000\ °\mathrm{C}$ (all in the same plot).

7.5 Kinetic friction

An object with a mass of 5.5 kg slides from rest down an inclined plane. The plane makes an angle of $\theta = 30°$ with the horizontal and is $s = 72$ m long. The speed of the object at the bottom of the plane is $v = 16.7\ \mathrm{m\ s^{-1}}$ and follows from

$$v = \sqrt{2g(\sin\theta - \mu\cos\theta)s},$$

where $g = 9.81\ \mathrm{m\ s^{-2}}$ and μ is the coefficient of kinetic friction between the plane and the object.

Tasks:
Write a structured and well-commented Fortran 90 program which uses Newton's numerical root finding method as described in section 4.5 to determine μ.

Program design:

1. The value of v is keyboard input and the maximum number of iterations is 20.
2. Use an initial value of $\mu = 0.5$ to start the root finding algorithm.

3. Terminate the calculations if $\Delta \equiv ||\mu_{i+1}| - |\mu_i|| < 10^{-5}$.
4. Use FUNCTIONs to determine $F(\mu)$ and $dF(\mu)/d\mu$.
5. For each iteration step, write Δ and μ to standard output.

7.6 Compton scattering

Suppose that x-rays of $E = 100$ keV energy are incident on a target and undergo so-called Compton (i.e. electron–photon) scattering. The scattering process can be described by the following formula,

$$\cos\phi = \frac{E^2 - E'^2 + K^2(1 + 2E_0/K)}{2EK\sqrt{1 + 2E_0/K}},$$

where $\phi = 73°$ is the angle of the recoiling electrons, E' is the energy of the scattered x-rays, $K = 2.5$ keV, and $E_0 = 511$ keV is the restmass of an electron.

Tasks:
Write a structured Fortran 90 program which uses the numerical root finding method as described in section 4.5 to determine E'.

Program design:
1. The value of E is keyboard input.
2. Limit the maximum number of iterations to 20.
3. Use an initial value of $E' = 10$ keV to start the root finding algorithm.
4. Terminate the calculations if $\Delta \equiv ||E'_{i+1}| - |E'_i|| < 10^{-5}$.
5. Use FUNCTIONs to determine $F(E')$ and $dF(E')/dE'$.
6. For each iteration step, write Δ and E' to standard output.

7.7 Radioactive decay

Given are three radioactive atomic nuclei, A, B, C, which decay according to the following radioactive decay chain:

$$A \longrightarrow B \longrightarrow C.$$

The decay is described by the following system of coupled differential equations,

$$dA/dt = -k_A A, \quad dB/dt = k_A A - k_B B, \quad dC/dt = k_B B, \tag{7.8}$$

where k_A and k_B are decay constants, and $A(t)$, $B(t)$, $C(t)$ are the number of nuclei of each species present. The differential equations are coupled, since each of the second and third of the them involves two of the dependent variables.

The initial values are $A(0) = A_0$, $B(0) = B_0$, and $C(0) = C_0$. The differential equations then have a unique solution, and that solution will depend on the parameters k_A and k_B and on the three initial values.

Tasks:
Write a structured and well-commented Fortran 90 program which solves the radioactive decay equations (7.8) using Euler's method as described in section 4.6.1.

Program design:

1. Compute solutions for $A(0) = 1000$, $B_0 = C_0 = 0$, $k_A = k_B = 0.1$, and a time step of $\Delta t = 0.25$ s.
2. Generate a plot which shows $A(t)$, $B(t)$, $C(t)$ for $0 \leqslant t \leqslant 50$ s.
3. Comment on the results for $A + B + C$. How are they connected with (7.8)?

7.8 Halley's comet

Halley's comet last reached perihelion (its point of closest approach to the Sun at the origin) on 9 February 1986. Its position and velocity components at this time were

$$\vec{\mathbf{r}}(0) \equiv (x(0),\, y(0),\, z(0)) = (0.325\,514,\, -0.459\,460,\, 0.166\,229)$$
$$\vec{\mathbf{v}}(0) \equiv \left(v_x(0),\, v_y(0),\, v_z(0)\right) = (-9.096\,111,\, -6.916\,686,\, -1.305\,721),$$

respectively, with position in AU (Astronomical Units, the unit of distance being equal to the major semi-axis of the Earth's orbit about the Sun). The time is measured in years. In this unit system, its three-dimensional equations of motion are as follows:

$$\frac{d^2x}{dt^2} = -\frac{\mu x}{r^3}, \qquad \frac{d^2y}{dt^2} = -\frac{\mu y}{r^3}, \qquad \frac{d^2z}{dt^2} = -\frac{\mu z}{r^3}, \tag{7.9}$$

where $\mu = 4\pi^2$, and $r = \sqrt{x^2 + y^2 + z^2}$.

Tasks:

Write a structured Fortran 90 program which solves (7.9) numerically via one or more of the methods described in sections 4.6.1–4.6.3 to find and illustrate the results graphically via a plot of the yz projection (i.e. z versus y) of the orbit of Halley's comet.

Use your numerical solution to determine Halley's maximum distance (at aphelion), the comet's period of revolution, and the time needed to return to perihelion. (Hint: plot $r(t)$.)

Using your results, determine the best estimate of the calendar date of the comet's next perihelion passage?

Program design:

1. Use the `Parameter` statement for π.
2. Use a time-step of $\Delta t = 0.001$ and have user input for the final time.
3. Have your program also generate plots for both the xy (i.e. y versus x) and xz (i.e. z versus x) projections.

Additional tasks:

Investigate your own comet

Lucky you! The night before your birthday in 1997 you set up your telescope on a nearby mountaintop. It was a clear night, and at 12:30 am you spotted a new comet. After repeating the observation on successive nights, you were able to calculate its solar system coordinates $\mathbf{r}_0 = (x(0), y(0), z(0))$ and its velocity vector

$\mathbf{v}_0 = (v_x(0), v_y(0), v_z(0))$ on that first night. Using this information, determine this comet's

- perihelion (point nearest Sun) and aphelion (farthest from Sun),
- its velocity at perihelion and at aphelion,
- its period of revolution about the Sun and,
- its next two dates of perihelion passage.

Using length-time units of AU and Earth years, the comet's equations of motion are given in equation (7.9) with $\mu = 4\pi^2$. For your personal comet, start with random initial position and velocity vectors with the same order of magnitude as those of Halley's comet. Repeat the random selection of initial position and velocity vectors, if necessary, until you obtain a nice-looking eccentric orbit that goes well outside the Earth's orbit (like real comets do).

7.9 Rocket equation

The equation that describes the motion of a rocket is given by the following differential equation:

$$\frac{dv}{dt} = \frac{R\,u_{\mathrm{ex}}}{m_i - Rt} - g. \tag{7.10}$$

This equation is called the rocket equation. The quantity $F_{\mathrm{th}} = Ru_{\mathrm{ex}}$ is the force exerted on the rocket by the exhausting fuel, and is called the thrust (R denotes the burn rate, u_{ex} is the speed at which the fuel is exhausted relative to the rocket). The payload of a rocket is defined as m_f/m_i, where m_f denotes the rocket's final mass, after all the fuel has been burned, and m_i is the rocket's initial mass. The quantity g denotes the gravitational acceleration (9.81 m s^{-2}). The Saturn-V rocket used in the Apollo moon-landing program had an initial mass of $m_i = 2.85 \times 10^6$ kg, a payload of 27%, a burn rate of 1.384×10^4 kg s^{-1}, and a thrust of 3.4×10^7 N.

Tasks

Using these values, write a Fortran 90 program that solves the rocket equation (7.10) numerically via one or more of the methods described in sections 4.6.1–4.6.3.

Program design:

1. Use the `PARAMETER` statement to assign a value to the gravitational acceleration in your program.
2. Use the `DATA` statement to assign values to the rocket data.
3. The temporal step size, dt, is to be read from the keyboard.
4. Compute the rocket's final mass, m_f, in a `FUNCTION` subprogram called RMASS_F.
5. Compute the rocket's burn time, given by $t_b = (m_i - m_f)/R$, in a `FUNCTION` subprogram called `BURNT`.
6. Print the values of u_{ex}, m_f, t_b on the terminal screen. The output should look as follows:

u_{ex} (km s^{-1})	m_f (tons)	t_b (sec)
.	.	.
.	.	.

7. Solve the rocket equation in a `SUBROUTINE` called `VELOCITY` and write the results to an external data file.
8. In the same subroutine, compute the exact value of the rocket's velocity at time t, which is given by

$$v(t) = -u_{ex} \ln(1 - Rt/m_i) - gt \tag{7.11}$$

   and save the results to another external data file.
9. Generate a plot that compares the numerical solution of the rocket equation (computed for time steps of 0.1, 1, 10, and 20 s) with the exact (i.e. analytical) solution given by equation (7.11).

7.10 Hydrostatic equilibrium and relativistic stars

Galaxies are filled with billions of so-called compact stars. Such objects are as massive as our Sun but have radii that are just around 10 km.[1] The densities inside compact stars are therefore 10–20 times higher than the density of atomic nuclei! In this assignment we will compute the mass–radius relationship of such stars, which follows from the first-order differential equation

$$\frac{dP(r)}{dr} = -\frac{[\epsilon + P(r)][4\pi r^3 P(r) + m(r)]}{r^2\left(1 - \dfrac{2m(r)}{r}\right)}, \tag{7.12}$$

where P, ϵ, and m denote the pressure, energy density, and mass of the mass distribution at a radial distance r from the center of the star, where we have used the geometrical units of $G = c = 1$.

Equation (7.12) follows from Albert Einstein's theory of general relativity and is known as the Tolman–Oppenheimer–Volkoff (TOV) equation as first derived in [1, 2]. The Newtonian limit of (7.12) is given by ($P \ll \epsilon$, $P \ll m$, $m/r \ll 1$)

$$\frac{dP(r)}{dr} = -\frac{\epsilon(r)\, m(r)}{r^2}, \tag{7.13}$$

and is known as the equation of classical hydrostatic equilibrium. This equation describes the pressure gradient inside of a spherically symmetric mass distribution. The mass contained in a spherical shell of radius r is given by

$$m(r) = 4\pi \int_0^r r'^2 \epsilon(r') dr'. \tag{7.14}$$

[1] The mass of our Sun is $M_\odot = 2 \times 10^{30}$ kg, its radius is around $R \sim 700\,000$ km.

The total mass M of the star is given by $M = m(R)$, where R denotes the stellar radius defined by $P(r = R) = 0$. The equations of hydrostatic equilibrium can be solved if the so-called equation of state (EoS) of the star is known (particle composition). In this assignment we will be studying stars made up of a relativistic gas of quarks. The EoS of such a system is extremely simple [3, 4],

$$P = (\epsilon - 4B)/3. \tag{7.15}$$

Here P and ϵ denote pressure and energy density in units of MeV fm^{-3}, and $B = 57$ MeV fm^{-3} is a constant.

Tasks

Write a Fortran 90 program that solves the coupled set of (7.12) and (7.14) numerically via one or more of the methods described in section 4.6.1–4.6.3 for given central energy densities $\epsilon_c \equiv \epsilon(r = 0)$ ranging from 4.2 B to 2000 B. The finite-difference representation of (7.12) and (7.14) is given by

$$\Delta m = 4\pi\epsilon r^2 \Delta r, \tag{7.16}$$

$$P(r + \Delta r) = P(r) - f(r)\Delta r, \tag{7.17}$$

where

$$f(r) \equiv \frac{(\epsilon + P)(4\pi r^3 P + m)\kappa}{r^2(1 - 2\, m\kappa/r)}. \tag{7.18}$$

The units in (7.16) to (7.18) are as follows: $[\epsilon] =$ MeV fm^{-3}, $[P] =$ MeV fm^{-3}, $[m] =$ MeV, $[r] =$ fm.

In the units of $G = c = 1$, the mass of the Sun is $M_\odot = 1.47$ km. On the other hand $M_\odot c^2 = M_\odot = 1.115\,829 \times 10^{60}$ MeV so that $\kappa \equiv 1.475 \times 10^{18}$ fm/$1.115\,829 \times 10^{60}$ MeV, which relates MeV to fm.

Program design:

1. Choose a step-size of $\Delta\epsilon_c = B/10$ and a radial step-size of $r = 10$ m.
2. Use the `Do While` construct for the range of central densities ($\Delta\epsilon_c$) and for the condition of $P(r) > 0$.
3. The stellar radius R (in km) and mass M (in units of the mass of the Sun, $M_\odot$) are to be written, for each central density ϵ_c, to an external data file.
4. Generate plots for $M/M_\odot$ as a function of R. Repeat the calculation for Newtonian stars (i.e. equation (7.13)) and show the outcome in the same plot. Comment on any differences.

Additional tasks:

Have your program also compute the gravitational redshift of light,

$$z = \left(1 - \frac{2\,M}{R}\right)^{-1/2} - 1,$$

as a function of mass M and write the results to an external data file.

Have your program generate a plot of z as a function of $M/M_\odot$. Repeat the calculation for Newtonian stars and show the outcome in the same plot. Comment on any differences.

7.11 Massive stars

In this homework, you will compute the properties of massive stars. We will assume that such stars are made of a gas of hydrogen atoms whose thermodynamic properties are described by the ideal gas equation of state, $PV = NkT$. For our purposes it is convenient to write the ideal gas equation of state as

$$P = \mu\ k\ T\ /\ m_{\rm H}, \tag{7.19}$$

where $k = 8.617 \times 10^{-11}$ MeV K^{-1} denotes the Boltzmann constant and $m_{\rm H} = 1.674 \times 10^{-27}$ kg is the mass of a hydrogen atom. Assume that the temperature is constant throughout the star (a very poor approximation) and is given by 5000 K. The mass density at the star's center is 100 g cm^{-3}. Finally, the pressure at the surface of such stars is typically 10^{-3} atm[2].

Tasks:
Rewrite the Fortran 90 program that you developed to compute the properties of compact stars (i.e. homework assignment 7.10) such that it computes the mass M (in units of $M_\odot$) and radius R (in km) of this star.

Program design:

1. Solve the hydrostatic equilibrium equations (i.e. (7.12) and (7.13)) in the Newtonian limit and determine by how much general relativity changes the Newtonian results for M and R.
2. Have your program generate a plot for the pressure profile $P(r)$ (in atm) and the particle number density profile $\rho(r) = P(r)/kT$ (in 1/cm^3) for this star.

7.12 Isothermal gas spheres

Poisson's equation for the gravitational potential Φ of a gas sphere is given by

$$\frac{\partial^2\Phi}{\partial r^2} + \frac{2}{r}\frac{\partial\Phi}{\partial r} = -a^2\, {\rm e}^{\Phi/K}, \tag{7.20}$$

where $a^2 \equiv 4\pi G\rho_0 = 2.91 \times 10^{-6}\,{\rm s}^{-1}$ and $K \equiv 10^{10}$ m^2 s^{-2}. The quantity $\rho_0 = 2.91 \times 10^3$ kg m^{-3} denotes the mass density at the center of the sphere. Its radial dependence is given by

$$\rho(r) = \rho_0 {\rm e}^{\Phi(r)/K}. \tag{7.21}$$

[2] 1 atm = 1.013×10^5 Pa, 1 Pa = 10^{-5} bar, 1 bar = 10^6 dyn cm^{-2}, 1 MeV fm^{-3} = 1.6022×10^{33} dyn cm^{-2}, 1 MeV fm^{-3} = 1.7827×10^{12} g cm^{-3}.

The initial conditions for the gravitational potential are $\Phi(r = 0) = 0$ and $\Phi'(r = 0) = 0$, where the prime denotes partial differential with respect to r. The finite difference solution of (7.20) reads

$$\Phi_i = 2\Phi_{i-1} - \Phi_{i-2} - \Delta r^2\left[\frac{2}{r_{i-1}}\frac{\Phi_{i-1} - \Phi_{i-2}}{\Delta r} + a^2 e^{\Phi_{i-1}/K}\right] \quad i = 2, 3, \ldots \tag{7.22}$$

with $\Phi_0 = 0$ and $\Phi_1 = 0$.

Tasks:

Write a structured and well-commented Fortran 90 program which solves (7.22) via one or more of the methods described in section 4.6.1–4.6.3 for $0 < r < r_{\mathrm{final}}$, where $r_{\mathrm{final}} = 8 \times 10^8$ km.

Program design:

1. Use a radial step size of $\Delta r = 5 \times 10^4$ km.
2. Use the `DO WHILE` construct for the radial integration. The initial conditions for Φ are $\Phi_0 = 0$ and $\Phi_1 = 0$.
3. The results for ρ as a function of r and $\log_{10}(\rho/\rho_0)$ as a function of $\log_{10}(r/\mathrm{km})$ are to be written to output files.
4. Illustrate your results for $\rho(r)$ and $\log_{10}(\rho/\rho_0)$ graphically (two separate plots).

7.13 Proton in constant electric and magnetic fields

The non-relativistic equation of motion of an electric charge q with mass m moving in a combined electric and magnetic field is given by

$$\frac{d\vec{v}}{dt} = \frac{q}{m}\vec{E} + \frac{q}{m}\vec{v} \times \vec{B}$$

The following conditions will produce a trochoidal motion in the x–y plane. Let

$$\vec{B} = B_z\vec{k}, \quad \vec{E} = E_y\vec{j}, \quad \vec{v}(0) = v_y(0)\vec{j}, \quad \vec{r}(0) = 0\vec{i} + 0\vec{j}.$$

Then the x and y components of the acceleration are given by

$$\frac{d^2x}{dt^2} = \frac{q}{m}\frac{dy}{dt}B_z, \tag{7.23}$$

$$\frac{d^2y}{dt^2} = \frac{q}{m}E_y - \frac{q}{m}\frac{dx}{dt}B_z. \tag{7.24}$$

Equations (7.23) and (7.24) can be written as a system of coupled first-order differential equations,

$$\dot{x} = v_x, \tag{7.25}$$

$$\dot{v}_x = \ddot{x}, \tag{7.26}$$

$$\dot{y} = v_y, \tag{7.27}$$

$$\dot{v}_y = \ddot{y}. \tag{7.28}$$

The input data and initial conditions for this problem are $m = 1.0 \times 10^{-27}$ kg, $q = 1.0 \times 10^{-19}$ C, $E_y = 1.5 \times 10^6$ V m^{-1}, $B_z = 0.1$ T, $x(0) = 0$, $y(0) = 0$, $v_x(0) = 0$, $v_y(0) = 1.0 \times 10^6$ m s^{-1}.

Tasks:
Write a structured Fortran 90 program which solves (7.25) through (7.28) numerically via one or more of the methods described in sections 4.6.1–4.6.3 for $0 < t < t_{\text{final}}$, where $t_{\text{final}} = 1.2 \times 10^{-6}$ s.

Program design:
1. Use a temporal step size of $\Delta t = 5.0 \times 10^{-10}$ s.
2. Assign numerical values to m, q, E_y, B_z $x(0)$, $y(0)$, $v_x(0)$, and $v_y(0)$ in a `SUBROUTINE` named `input_data`.
3. Define a `SUBROUTINE` named `write` which writes the values assigned to m, q, E_y, B_z $x(0)$, $y(0)$, $v_x(0)$, and $v_y(0)$ back to standard output.
4. The differential equations (7.25) through (7.28) are to be solved in a `SUBROUTINE` named `diffeqs`.
5. The proton's position, i.e. y as a function of x (given in meters), is to be written to an output file.
6. Have your program generate a plot which illustrates the proton's position graphically.

7.14 Square voltage pulse applied to a RC circuit

A resistor–capacitor circuit (RC circuit [5]) is an electric circuit composed of resistors (R) and capacitors (C) driven by a voltage or current source. An RC circuit consisting of only one resistor and one capacitor is shown in figure 7.3.

A square voltage pulse with a time dependence given by

$$U(t) = \begin{cases} 0, & \text{if } t < t_1 \\ U_0 = 6\text{ V}, & \text{if } t_1 \leqslant t \leqslant t_2 \\ 0, & \text{if } t_2 < t \end{cases} \tag{7.29}$$

is applied to the RC circuit shown in figure 7.3. The circuit current $\dot{q}$ $(=I)$ is given by the equation

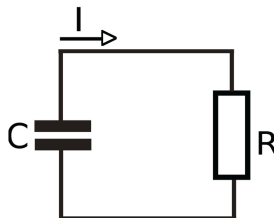


Figure 7.3. Illustration of a simple RC circuit.

$$R\,\frac{dq}{dt} + \frac{1}{C}\,q = U(t), \tag{7.30}$$

where $R = 100\ \Omega$ and $C = 4.7 \times 10^{-5}$ A s V^{-1}. The initial condition is $q(0) = 0$.

Tasks

Write a complete Fortran 90 program which solves (7.30) (i.e. computes $q(t)$ and $I(t)$) numerically via one or more of the methods described in sections 4.6.1–4.6.3 for times $0 < t < t_{\text{final}}$, where $t_{\text{final}} = 4RC$, $t_1 = 0$, and $t_2 = 2RC$.

Program design

1. There is NO input from keyboard.
2. Use $\Delta t = RC/200$ for the incremental time step.
3. The results for $q(t)$ and $I(t)$ are to be written to an external data file.
4. The mathematical solution of (7.30) for the current $I(t)$ is given by

$$\bar{I}(t) = \frac{U_0}{R}\left(\mathrm{e}^{-(t-t_1)/(RC)}\,\Theta(t - t_1) - \mathrm{e}^{-(t-t_2)/(RC)}\,\Theta(t - t_2)\right), \tag{7.31}$$

   where Θ denotes the Heaviside step function given by

$$\Theta(x) = \begin{cases} 0, & \text{if } x < 0 \\ 1, & \text{if } x \geqslant 0 \end{cases}$$

   Design your code such that $\bar{I}(t)$ is computed for $0 < t < t_{\text{final}}$. The result is to be written to an external data file.
5. Generate a plot which shows $I(t)$ and $\bar{I}(t)$ for $0 < t < t_{\text{final}}$.

7.15 Mutual inductance of two coils

Mutual inductance [5] is the basic operating principal of the transformer, motors, generators, and any other electrical component that interacts with another magnetic field. Mutual induction is defined as the current flowing in one coil that induces a current in an adjacent coil. An example is shown in figure 7.4, where current I_1 flowing in coil L_1 caused a current I_2. The differential equations which describes the flow of the electric currents are given by

$$L_1\,\dot{I}_1 + R_1\,I_1 - L_{12}\,\dot{I}_2 = U, \tag{7.32}$$

$$L_2\,\dot{I}_2 + R_2\,I_2 - L_{12}\,\dot{I}_1 = 0, \tag{7.33}$$

where $L_1 = 1.6$ H, $L_2 = 0.9$ H, $L_{12} = 0.72$ H, $R_1 = 48\ \Omega$, $R_2 = 27\ \Omega$, and $U = 240$ V[3].

Tasks:

Write a structured Fortran 90 program which solves (7.32) and (7.33) numerically via one or more of the methods described in sections 4.6.1–4.6.3 for times

[3] H = Ω s, F = s Ω^{-1}.

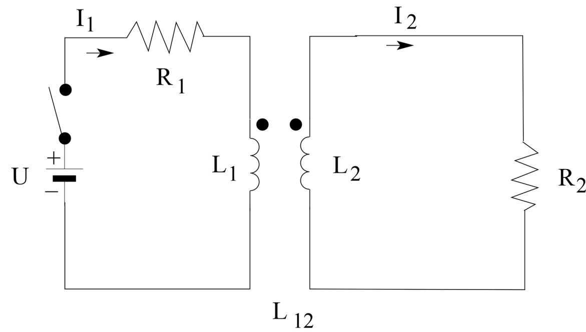


Figure 7.4. Mutual inductance between coils.

$0 \leqslant t \leqslant t_{\text{final}}$, where $t_{\text{final}} = 0.25$ s. The initial conditions for the electric currents are $I_1(0) = I_2(0) = 0$ A.

Program design:

1. Choose $\Delta t = t_{\text{final}}/250$ for the temporal step size.
2. The analytic solutions of (7.32) and (7.33) are given by

$$\bar{I}_1(t) = 5 - 2.5\mathrm{e}^{-75t} - 2.5\mathrm{e}^{-75t/4} \tag{7.34}$$

$$\bar{I}_2(t) = -3.33\mathrm{e}^{-75t} + 3.33\mathrm{e}^{-75t/4}. \tag{7.35}$$

   Use FUNCTIONs to compute the analytic solutions given by (7.34) and (7.35).
3. Use the Fortran 90 module feature to assign values to L_1, L_{12}, L_2, R_1, R_2, and U.
4. The initial value of $\bar{I}_1$ and $\bar{I}_2$ are keyboard input.
5. Generate a plot which shows the numerical as well as the analytic results for $I_1(t)$ and $I_2(t)$ for $0 \leqslant t \leqslant t_{\text{final}}$.

References

[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)

IOP Concise Physics

Introduction to Computational Physics for Undergraduates

Omair Zubairi and Fridolin Weber

Appendix A

Summary of Fortran features

This overview of Fortran 90/95 features is presented as a series of tables that illustrate the syntax and abilities of Fortran 90/95. Comparisons are made to similar features in the Fortran 77 language. The tables (A.1–A.3) show that Fortran 90/95 has significant improvements over Fortran 77 and matches or exceeds newer software capabilities found in C++ and Matlab for dynamic memory management, user defined data structures, matrix operations, operator definition and overloading,

Table A.1. This table shows array operations in programming constructs. Lower-case letters denote scalar elements or arrays, upper-case letters denote matrices or scalar elements of matrices.

Description	Equation	F90/F95 operation
Scalar plus scalar	$c = a \pm b$	$c = a \pm b$
Element plus scalar	$c_{jk} = a_{jk} \pm b$	$c = a \pm b$
Element plus element	$c_{jk} = a_{jk} \pm b_{jk}$	$c = a \pm b$
Scalar times scalar	$c = a \times b$	$c = a*b$
Element times scalar	$c_{jk} = a_{jk} \times b$	$c = a*b$
Element times element	$c_{jk} = a_{jk} \times b_{jk}$	$c = a*b$
Scalar divide scalar	$c = a/b$	$c = a/b$
Scalar divide element	$c_{jk} = a_{jk}/b$	$c = a/b$
Element divide element	$c_{jk} = a_{jk}/b_{jk}$	$c = a/b$
Scalar power scalar	$c = a^b$	$c = a**b$
Element power scalar	$c_{jk} = a_{jk}^b$	$c = a**b$
Element power element	$c_{jk} = a_{jk}^{b_{jk}}$	$c = a**b$
Matrix transpose	$C_{kj} = A_{jk}$	$C =$ `transpose`(A)
Matrix times matrix	$C_{ij} = \sum_k A_{ik} B_{kj}$	$C =$ `matmul`(A, B)
Vector dot vector	$c = \sum_k A_k B_k$	$c =$ `sum`$(A*B)$
		$c =$ `dot_product`(A, B)

doi:10.1088/978-1-6817-4896-2ch8

© Morgan & Claypool Publishers 2018

Table A.2. Fortran features to include intrinsic data types, relational operators, and flow control statement.

Description	F77	F90/F95
Comment syntax	`C,*`	`!`
byte	`character`	`character::`
integer	`integer`	`integer::`
single precision	`real`	`real::`
double precision	`double precision`	`real*8::`
complex	`complex`	`complex::`
argument	`parameter`	`parameter::`
pointer	–	`pointer::`
structure	–	`type::`
Equal to	`.EQ.`	`==`
Not equal to	`.NE.`	`/=`
Less than	`.LT.`	`<`
Less or equal	`.LE.`	`<=`
Greater than	`.GT.`	`>`
Greater or equal	`.GE.`	`>=`
Logical NOT	`.NOT.`	`.NOT.`
Logical AND	`.AND.`	`.AND.`
Logical inclusive OR	`.OR.`	`.OR.`
Logical exclusive OR	`.XOR.`	`.XOR.`
Logical equivalent	`.EQV.`	`.EQV.`
Logical not equivalent	`.NEQV.`	`.NEQV.`
Conditionally execute statements	`if`	`if`
	`end if`	`end if`
Loop a specific number of times	`do # k=1,n`	`do k=1,n`
	`# continue`	`end do`
Loop an indefinite number of times	–	`do while`
	–	`end do`
Terminate and exit loop	`go to`	`exit`
Skip a cycle of loop	`go to`	`cycle`
Display message and abort	`stop`	`stop`
Return to invoking function	`return`	`return`
Conditional array action	–	`where`
Conditional alternative statements	`else`	`else`
	`elseif`	`elseif`
Conditional array alternatives	–	`elsewhere`
Conditional case selections	`if`	`select case`
	`end if`	`end select`

intrinsics for vector and parallel processors, and the basic requirements for object-oriented programming. They are intended to serve as a condense quick reference guide for programming in Fortran 90/95 and for understanding programs developed by others.

Table A.3. Overview of Fortran 90 intrinsic functions. The names of the arguments specify their type (i.e. X = Real, DX = Double precision, IX = Integer, Z = Complex).

F90	Function type	Definition
SQRT(X)	Real	$\sqrt{X}$
DSQR(DX)	Double precision	$\sqrt{DX}$
ABS(X)	Real	$\|X\|$
EXP(X)	Real	e^X
DEXP(DX)	Double precision	e^{DX}
LOG(X)	Real	$\log_e X$
LOG10(X)	Real	$\log_{10} X$
IFIX(X)	Integer	Truncate X to an integer
AINT(X)	Real	Round number
NINT(X)	Real	Round X to an integer
FLOAT(X)	Real	Converts IX to real value
CEILING(X)	Real	Smallest integer $>X$
FLOOR(X)	Real	Largest integer $<X$
MOD(X,Y)	Real	Division remainder
CONJ(Z)	Real	Complex conjugate
IMAG(Z)	Real	Imaginary part
DBLE(X)	Double precision	Convert X to double precision
AMAX1(X,Y,...)	Real	Maximum of $(X, Y, \ldots)$
AMAX0(IX,IY,...)	Real	Maximum of $(IX, IY, \ldots)$
AMIN0(IX,IY,...)	Real	Minimum of $(IX, IY, \ldots)$
AMIN1(X,Y,...)	Real	Minimum of $(X, Y, \ldots)$
MIN0(IX,IY,...)	Integer	Minimum of $(IX, IY, \ldots)$
SIN(X)	Real	$\sin(X)$
COS(X)	Real	$\cos(X)$
TAN(X)	Real	$\tan(X)$
ASIN(X)	Real	$\arcsin(X)$
ACOS(X)	Real	$\arccos(X)$
ATAN(X)	Real	$\arctan(X)$
SINH(X)	Real	$\sinh(X)$
COSH(X)	Real	$\cosh(X)$
TANH(X)	Real	$\tanh(X)$

IOP Concise Physics

Introduction to Computational Physics for Undergraduates

Omair Zubairi and Fridolin Weber

Appendix B

Plotting using Python

There are many plotting programs and software available to use. In this appendix, we show how to plot data using the Python programming language. Plotting data using Python is very useful and versatile and can be performed on multiple platforms The file extension for a Python program is `.py` (i.e. `filename.py`). To compile and run a Python program, simply type

```
> python filename.py
```

Figure B.1 shows a simple graph of some numerical data. The sample Python code which produced figure B.1 is shown below for your reference.

```
#! /usr/bin/python3

import numpy as np
import matplotlib.pyplot as plt

# import data:
fx = np.loadtxt('datafile.dat')

# reading in two columns:
plt.plot(fx[:,0],fx[:,1])

# setting horizontal axis
plt.xlim((0,5))

# setting vertical axis
plt.ylim((0,30))
```

doi:10.1088/978-1-6817-4896-2ch9 © Morgan & Claypool Publishers 2018

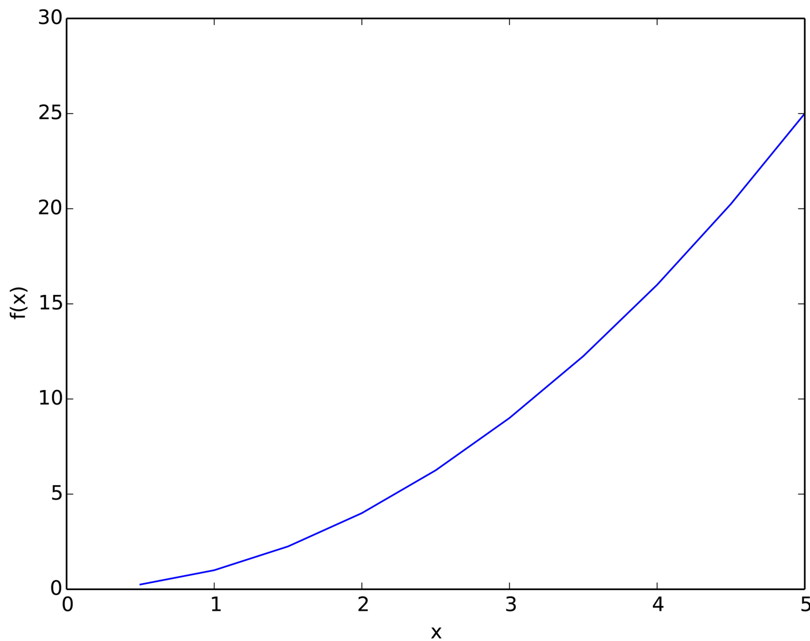


Figure B.1. Plot of the function $f(x) = x^2$.

```
# label axis
plt.xlabel('x')
plt.ylabel('f(x)')

# saving plot into a .pdf format
plt.savefig('graph.pdf')
plt.clf()
```

Note that the symobl '#' is used for comments. The sample code above gives the basics of plotting data; however, much more can be done using Python such as configuring axis, including a title, insert special symbols, etc. For example, in figure B.2, the axes are configured differently with more increments, there is a title for the graph, and we have inserted a special character on the vertical axis.

The sample code below gives a more detailed description on how to reproduce figure B.2.

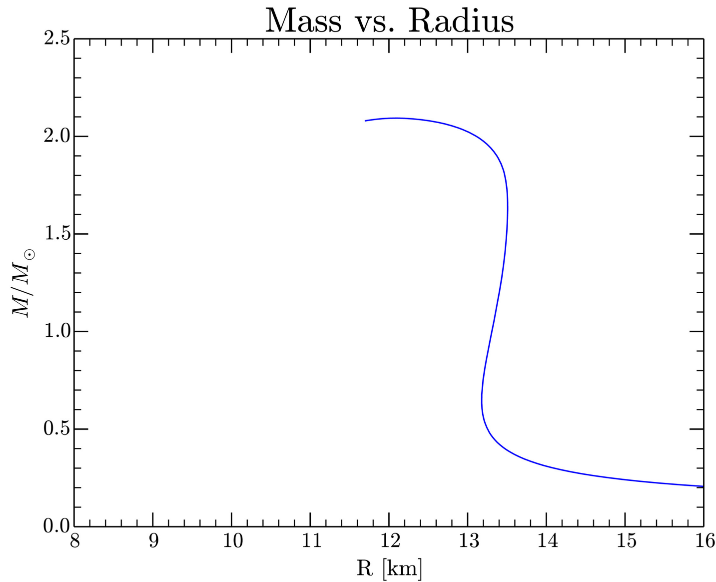


Figure B.2. Mass–radius plot for neutron stars.

```
#! /usr/bin/python

import numpy as np
import matplotlib.pyplot as plt

#import data
mr = np.loadtxt('mass_radius.dat')

#Note: The .dat file has 3 columns -- e_c, mass, radius
#but the columns are e_c = 0, mass = 1, radius = 2

#specify which columns of data are being imported
plt.plot(mr[:,2],mr[:,1])

#Note: ([x],[y])->([radius], [mass]) = ([:,2], [:,1])

#title of plot
plt.title('Mass vs. Radius', fontname='cmr10', fontsize=25)

#configure ticks:
plt.minorticks_on()
plt.tick_params(axis='both',which='minor',length=5,width=0.75,labelsize=15)
plt.tick_params(axis='both',which='major',length=10,width=1.0,labelsize=15)

#configure x-axis
plt.xlim((8,16))                          #range of x-values
plt.xlabel(r'R [km]', fontname='cmr10', fontsize=16)
plt.xticks(fontname='cmr10', fontsize=15, size=15)
```

```
#configure y-axis
plt.ylabel(r'$M/M_\odot$', fontname='cmr10', fontsize=16)
plt.yticks(fontname='cmr10', fontsize=15, size=15)

#save your plot
plt.savefig('mr.pdf')
plt.savefig('mr.eps')
plt.clf()
```

IOP Concise Physics

Introduction to Computational Physics for Undergraduates

Omair Zubairi and Fridolin Weber

Appendix C

Fortran 90 sample program illustrating good programming

The following is a sample program which illustrates good Fortran 90 programming.

```
!* module
   module constants
   implicit none
   real, parameter :: pi=acos(-1.)     ! define pi
   real, parameter :: k_B =8.3145      ! k_B in joule/mol/K
   real, parameter :: m_N 2=28.        ! Molecular mass of N2 in g/mol
   real, parameter :: m_02=32.         ! Molecular mass of O2 g/mol
   real, parameter :: m_Ar=40.         ! Molecular mass of Ar g/mol
   end module constants

!* main program
   program boltzmann
!*
! Calculate the fraction of molecules in a thermally equilibrated gas
! (uniform temperature T) whose speed is less than a given speed v,
! given by the expression
!                                   v
! f(v,T) = 4*pi*(m/2/pi/k/T)^1.5 * I dv' exp(-m*v'^2/2/xk/T)*v'^2
!                                   0
!
! Performing the variable transformation v^2/kappa^2=y^2 with
! kappa=(2kT/m)^1/2 leads for f(v,T) to
!                              v/kappa
```

doi:10.1088/978-1-6817-4896-2ch10 © Morgan & Claypool Publishers 2018

```
!  f(v,t) = 4*pi/pi^3/2 * I dy y^2 exp(-y^2)
!                              0
! Given:
!  v    = Speed (m/sec)
!  m_N2 = Mole mass of nitrogen = 28 g/mol
!  m_O2 = Mole mass of oxygen = 32 g/mol
!  m_Ar = Mole mass of argon = 40 g/mol
!        (Air consists of 78%N2, 21% 02, 1% Ar)
!  k_B  = Boltzmann constant = 8.319 Joule/mol/K
!  T    = Temperature (K)

      use constants

      implicit none
      real :: nint, m, Tc, v_max, h_v, T, kappa, y_a, y_b, h_y
      real :: a, b, sum, y_k, f, fvT
      integer :: k, n

                                  ! Standard input from keyboard
      write(*,*) 'Compute fraction of molecules whose speeds v < v_max:'
      write(*,*) 'Enter temperature in Celsius: '
      read(*,*) Tc
      write(*,*) 'Enter upper velocity limit in m/sec: '
      read(*,*) v_max
      write(*,*) 'Enter integration step size in m/sec (e.g. 1.): '
      read(*,*) h_v

      T   = 273.15 + Tc               ! Compute absolute temperature (in K)
      m   = (m_N2*0.78 + m_02*0.21 + m_Ar*0.01) / 1000. ! Masses in kg/mol
      kappa = sqrt(2.*k_B*T/m)

      a=0.
      b=v_max
      y_a  = a/kappa
      y_b  = b/kappa
      h_y  = h_v/kappa
      nint = (y_b-y_a)/h_y

      n   = ifix(nint)


      sum = 0.
Boltzmann_integral: do k=1, n-1
```

```
                        y_k= y_a+h_y*float(k)
                        sum = sum + f(y_k)
                     end do Boltzmann_integral
                     fvT = h_y * (f(y_a)+f(y_b)+2.*sum) / 2.
                     fvT = 4.*pi * fvT / sqrt(pi)**3
! Output the result
   write(*,*) 'Results: '
   write(*,*) '   Average molecular mass (g/mol):  ', m*1000.
   write(*,*) '   Temperatuer of gas (Celsius):    ', Tc
   write(*,*) '   Velocity (m/sec)                 ', v_max
   write (*,20) fvT*100.
20  format('  f(v,T) (in %)                        ',3x,f6.2)


   stop
       end program boltzmann

!* function sub-program
   function f(y)
!*
   implicit none
   real :: f, y, y2

   y2 = y*y
   f = y2*exp(-y2)

   return
     end function f
```