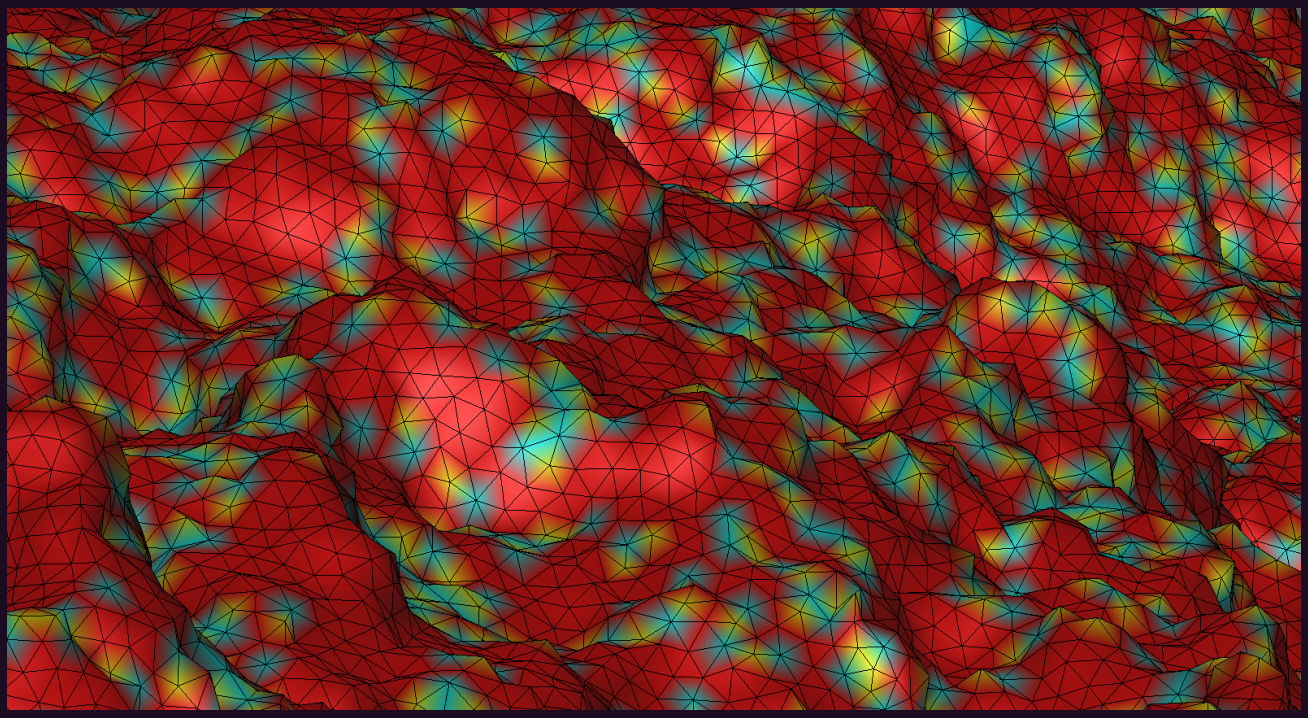
# *Computational Physics*



*Konstantinos N. Anagnostopoulos*

# COMPUTATIONAL PHYSICS

A Practical Introduction to Computational Physics
and Scientific Computing

Athens, 2014

KONSTANTINOS N. ANAGNOSTOPOULOS
*National Technical University of Athens*

National Technical University of Athens

PUBLISHED BY KONSTANTINOS N. ANAGNOSTOPOULOS
Physics Department, National Technical University of Athens, Zografou Campus, 15780 Zografou, Greece
konstant@mail.ntua.gr, www.physics.ntua.gr/~konstant/
and the NATIONAL TECHNICAL UNIVERSITY OF ATHENS

Book Website:
www.physics.ntua.gr/~konstant/ComputationalPhysics

©Konstantinos N. Anagnostopoulos 2014

First Published 2014
Version[1] 1.0.20140827051600

Cover: Design by K.N. Anagnostopoulos. The front cover picture is a snapshot taken during Monte Carlo simulations of hexatic membranes. Work done with Mark J. Bowick. Relevant video at youtu.be/Erc7Q6YXfLk

This book and its cover(s) are subject to copyright. They are licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit
creativecommons.org/licenses/by-sa/4.0/

This is a human-readable summary of (and not a substitute for) the license, see
creativecommons.org/licenses/by-sa/4.0/legalcode

You are free to:

- Share — copy and redistribute the material in any medium or format
- Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

Notices:
You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.
No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

The book is accompanied by software available at the book's website. All the software, unless the copyright does not belong to the author, is open source, covered by the GNU public license, see www.gnu.org/licenses/. This is explicitly mentioned at the end of the respective source files.

ISBN 978-1-312-31812-0 (lulu.com)

ISBN 978-1-312-46441-4 (lulu.com, vol. I)

ISBN 978-1-312-46488-9 (lulu.com, vol. II)

[1]The first number is the major version, corresponding to an "edition" of a conventional book. Versions differing by major numbers have been altered substantially. Chapter numbers and page references are not guaranteed to match between different versions. The second number is the minor version. Versions differing by a minor version may have serious errors/typos corrected and/or substantial text modifications. Versions differing by only the last number may have minor typos corrected, added references etc. When reporting errors, please mention the version number you are referring to.

# Contents

**Foreword** **vii**

**1 The Computer** **1**
1.1 The Operating System . . . . . 3
1.1.1 Filesystem . . . . . 3
1.1.2 Commands . . . . . 11
1.1.3 Looking for Help . . . . . 14
1.2 Text Processing Tools – Filters . . . . . 16
1.3 Programming with Emacs . . . . . 21
1.3.1 Calling Emacs . . . . . 21
1.3.2 Interacting with Emacs . . . . . 23
1.3.3 Basic Editing . . . . . 24
1.3.4 Cut and Paste . . . . . 27
1.3.5 Windows . . . . . 29
1.3.6 Files and Buffers . . . . . 30
1.3.7 Modes . . . . . 31
1.3.8 Emacs Help . . . . . 32
1.3.9 Emacs Customization . . . . . 34
1.4 The Fortran Programming Language . . . . . 35
1.4.1 The Foundation . . . . . 35
1.4.2 Details . . . . . 45
1.4.3 Arrays . . . . . 50
1.5 Gnuplot . . . . . 57
1.6 Shell Scripting . . . . . 62

**2 Kinematics** **77**
2.1 Motion on the Plane . . . . . 77
2.1.1 Plotting Data . . . . . 85
2.1.2 More Examples . . . . . 88
2.2 Motion in Space . . . . . 99
2.3 Trapped in a Box . . . . . 107

2.3.1 The One Dimensional Box . . . . . 108
2.3.2 Errors . . . . . 114
2.3.3 The Two Dimensional Box . . . . . 117
2.4 Applications . . . . . 121
2.5 Problems . . . . . 140

**3 Logistic Map 145**
3.1 Introduction . . . . . 145
3.2 Fixed Points and $2^n$ Cycles . . . . . 147
3.3 Bifurcation Diagrams . . . . . 154
3.4 The Newton-Raphson Method . . . . . 157
3.5 Calculation of the Bifurcation Points . . . . . 163
3.6 Liapunov Exponents . . . . . 167
3.7 Problems . . . . . 180

**4 Motion of a Particle 191**
4.1 Numerical Integration of Newton's Equations . . . . . 191
4.2 Prelude: Euler Methods . . . . . 192
4.3 Runge–Kutta Methods . . . . . 204
4.3.1 A Program for the 4th Order Runge–Kutta . . . . . 208
4.4 Comparison of the Methods . . . . . 212
4.5 The Forced Damped Oscillator . . . . . 215
4.6 The Forced Damped Pendulum . . . . . 222
4.7 Appendix: On the Euler–Verlet Method . . . . . 229
4.8 Appendix: 2nd order Runge–Kutta Method . . . . . 233
4.9 Problems . . . . . 236

**5 Planar Motion 241**
5.1 Runge–Kutta for Planar Motion . . . . . 241
5.2 Projectile Motion . . . . . 246
5.3 Planetary Motion . . . . . 252
5.4 Scattering . . . . . 256
5.4.1 Rutherford Scattering . . . . . 260
5.4.2 More Scattering Potentials . . . . . 267
5.5 More Particles . . . . . 270
5.6 Problems . . . . . 281

**6 Motion in Space 285**
6.1 Adaptive Stepsize Control for Runge–Kutta Methods . . . . 286
6.2 Motion of a Particle in an EM Field . . . . . 295
6.3 Relativistic Motion . . . . . 296

6.4 Problems . . . . . 307

**7 Electrostatics 311**
7.1 Electrostatic Field of Point Charges . . . . . 311
7.2 The Program – Appetizer and ... Desert . . . . . 314
7.3 The Program – Main Dish . . . . . 323
7.4 The Program - Conclusion . . . . . 329
7.5 Electrostatic Field in the Vacuum . . . . . 334
7.6 Results . . . . . 341
7.7 Poisson Equation . . . . . 342
7.8 Problems . . . . . 349

**8 Diffusion Equation 353**
8.1 Introduction . . . . . 353
8.2 Heat Conduction in a Thin Rod . . . . . 355
8.3 Discretization . . . . . 356
8.4 The Program . . . . . 358
8.5 Results . . . . . 361
8.6 Diffusion on the Circle . . . . . 363
8.7 Analysis . . . . . 366
8.8 Problems . . . . . 370

**9 The Anharmonic Oscillator 373**
9.1 Introduction . . . . . 373
9.2 Calculation of the Eigenvalues of $H_{nm}(\lambda)$ . . . . . 375
9.3 Results . . . . . 384
9.4 The Double Well Potential . . . . . 390
9.5 Problems . . . . . 398

**10 Time Independent Schrödinger Equation 401**
10.1 Introduction . . . . . 401
10.2 The Infinite Potential Well . . . . . 404
10.3 Bound States . . . . . 415
10.4 Measurements . . . . . 424
10.5 The Anharmonic Oscillator - Again... . . . . 431
10.6 The Lennard–Jones Potential . . . . . 435
10.7 Problems . . . . . 438

**11 The Random Walker 443**
11.1 (Pseudo)Random Numbers . . . . . 444
11.2 Using Pseudorandom Number Generators . . . . . 456

11.3 Random Walks . . . . . 463
11.4 Problems . . . . . 471

**12 Monte Carlo Simulations** **475**
12.1 Statistical Physics . . . . . 476
12.2 Entropy . . . . . 479
12.3 Fluctuations . . . . . 483
12.4 Correlation Functions . . . . . 485
12.5 Sampling . . . . . 487
12.5.1 Simple Sampling . . . . . 488
12.5.2 Importance Sampling . . . . . 489
12.6 Markov Processes . . . . . 490
12.7 Detailed Balance Condition . . . . . 491
12.8 Problems . . . . . 493

**13 Simulation of the $d = 2$ Ising Model** **495**
13.1 The Ising Model . . . . . 495
13.2 Metropolis . . . . . 501
13.3 Implementation . . . . . 504
13.3.1 The Program . . . . . 509
13.3.2 Towards a Convenient User Interface . . . . . 515
13.4 Thermalization . . . . . 527
13.5 Autocorrelations . . . . . 529
13.6 Statistical Errors . . . . . 536
13.6.1 Errors of Independent Measurements . . . . . 537
13.6.2 Jackknife . . . . . 540
13.6.3 Bootstrap . . . . . 542
13.7 Appendix: Autocorrelation Function . . . . . 543
13.8 Appendix: Error Analysis . . . . . 551
13.8.1 The Jackknife Method . . . . . 551
13.8.2 The Bootstrap Method . . . . . 555
13.8.3 Comparing the Methods . . . . . 558
13.9 Problems . . . . . 563

**14 Critical Exponents** **573**
14.1 Critical Slowing Down . . . . . 575
14.2 Wolff Cluster Algorithm . . . . . 576
14.3 Implementation . . . . . 583
14.3.1 The Program . . . . . 585
14.4 Production . . . . . 590
14.5 Data Analysis . . . . . 593

14.6 Autocorrelation Times . . . . . 601
14.7 Temperature Scaling . . . . . 604
14.8 Finite Size Scaling . . . . . 610
14.9 Calculation of $\beta_c$ . . . . . 612
14.10 Studying Scaling with Collapse . . . . . 617
14.11 Binder Cumulant . . . . . 627
14.12 Appendix: Scaling . . . . . 630
14.12.1 Binder Cumulant . . . . . 630
14.12.2 Scaling . . . . . 636
14.12.3 Finite Size Scaling . . . . . 638
14.13 Appendix: Critical Exponents . . . . . 641
14.13.1 Definitions . . . . . 641
14.13.2 Hyperscaling Relations . . . . . 642
14.14 Problems . . . . . 642

**Bibliography** **646**

**Index** **654**

This book has been written assuming that the reader **executes all the commands presented in the text and follows all the instructions at the same time.** If this advice is neglected, then the book will be of little help and some parts of the text may seem incomprehensible.

The book's website is at
`http://www.physics.ntua.gr/~konstant/ComputationalPhysics/`
From there, you can can download the *accompanying software*, which contains, among other things, *all* the programs presented in the book.

**Some conventions**: Text using the font shown below refers to commands given using a shell (the "command line"), input and output of programs, code written in Fortran (or any other programming language), as well as to names of files and programs:

```
> echo Hello world
Hello world
```

When a line *starts* with the prompt

```
>
```

then the text that follows is a command, which can be given from the command line of a terminal. The second line, `Hello World`, is the *output* of the command.

The contents of a file with Fortran code is listed below:

```
program add
 z = 1.0
 y = 2.0
 x = z + y
 print *, x
end program add
```

What you need in order to work on your PC:

- An operating system of the GNU/Linux family and its basic tools.
- A Fortran compiler. The `gfortran` compiler is freely available for all major operating systems under an open source license at `http://www.gfortran.org`.
- An advanced text editor, suitable for editing code in several programming languages, like Emacs[2].
- A good plotting program, suitable for data analysis, like `gnuplot`[3].
- The shell `tcsh`[4].
- The programs `awk`[5], `grep`, `sort`, `cat`, `head`, `tail`, `less`. Make sure that they are available in your computer environment.

If you have installed a GNU/Linux distribution on your computer, all of the above can be installed easily. For example, in a Debian like distribution (Ubuntu, ...) the commands

```
> sudo apt-get install tcsh emacs gnuplot-x11 gnuplot-doc
> sudo apt-get install gfortran gawk gawk-doc binutils
> sudo apt-get install manpages-dev coreutils liblapack3
```

install all the necessary tools.

If you don't wish to install GNU/Linux on your computer, you can try the following:

- Boot your computer using a usb/DVD live GNU/Linux, like Ubuntu[6]. This will not make any permanent changes in your hard drive but it will start and run slower. On the other hand, you may save all your computing environment and documents and use it on any computer you like.
- Install `Cygwin`[7] in your Microsoft Windows. It is a very good solution for Microsoft-addicted users. If you choose the *full* installation, then you will find all the tools needed in this book.

[2] `http://www.gnu.org/software/emacs/`
[3] `http://www.gnuplot.info`
[4] `http://www.tcsh.org`
[5] `http://www.gnu.org/software/gawk`
[6] `http://www.ubuntu.com`
[7] `http://www.cygwin.com`

- Mac OS X is based on Unix. It is possible to install all the software needed in this book and follow the material as presented. Search the internet for instructions, e.g. google "gfortran for Mac", "emacs for Mac", "tcsh for Mac", etc.

# Foreword

This book is the culmination of my ten years' experience in teaching three introductory, *undergraduate level*, scientific computing/computational physics classes at the National Technical University of Athens. It is suitable mostly for junior or senior level science courses, but I am currently teaching its first chapters to sophomores without a problem. A two semester course can easily cover all the material in the book, including lab sessions for practicing.

Why another book in computational physics? Well, when I started teaching those classes there was no bibliography available in Greek, so I was compelled to write lecture notes for my students. Soon, I realized that my students, majoring in physics or applied mathematics, were having a hard time with the technical details of programming and computing, rather than with the physics concepts. I had to take them slowly by the hand through the "howto" of computing, something that is reflected in the philosophy of this book. Hoping that this could be useful to a wider audience, I decided to translate these notes in English and put them in an order and structure that would turn them into "a book".

I also decided to make the book freely available on the web. I was partly motivated by my anger caused by the increase of academic (e)book prices to ridiculous levels during times of plummeting publishing costs. Publishers play a diminishing role in academic publishing. They get an almost ready-made manuscript in electronic form by the author. They need to take no serious investment risk on an edition, thanks to print-on-demand capabilities. They have virtually zero cost ebook publishing. Moreover, online bookstores have decreased costs quite a lot. Academic books need no advertisement budget, their success is due to their academic reputation. I don't see all of these reflected on reduced book prices, quite the contrary, I'm afraid.

My main motivation, however, is the freedom that independent publishing would give me in improving, expanding and changing the book in the future. It is great to have no length restrictions for the presenta-

tion of the material, as well as not having to report to a publisher. The reader/instructor that finds the book long, can read/print the portion of the book that she finds useful for her.

This is *not* a reference book. It uses some interesting, I hope, physics problems in order to introduce the student to the fundamentals of solving a scientific problem numerically. At the same time, it keeps an eye in the direction of advanced and high performance scientific computing. The reader should follow the instructions given in each chapter, since the book *teaches by example*. Several skills are taught through the solution of a particular problem. My lectures take place in a (large) computer lab, where the students are simultaneously doing what I am doing (and more). The program that I am editing and the commands that I am executing are shown on a large screen, displaying my computer monitor and actions live. The book provides no systematic teaching of a programming language or a particular tool. A very basic introduction is given in the first chapter and then the reader learns whatever is necessary for the solution of her problem. There is more than one way to do it[8] and the problems can be solved by following a basic or a fancy way, depending on the student's computational literacy. The book provides the necessary tools for both. A bibliography is provided at the end of the book, so that the missing pieces of a puzzle can be sought in the literature.

This is also *not* a computational physics playground. Of course I hope that the reader will have fun *doing* what is in the book, but my goal is to provide an experience that will set the solid foundation for her becoming a high performance computing, number crunching, heavy duty data analysis expert in the future. This is why the programming language of the core numerical algorithms has been chosen to be Fortran, a highly optimized, scientifically oriented, programming language. The computer environment is set in a Unix family operating system, enriched by all the powerful GNU tools provided by the FSF[9]. These tools are indispensable in the complicated data manipulation needed in scientific research, which requires flexibility and imagination. Of course, Fortran is not the best choice for heavy duty object oriented programming, and is not optimal for interacting with the operating system. The philosophy[10]

[8] A Perl moto!

[9] Free Software Foundation, `www.fsf.org`.

[10] Java and C++ have been popular choices in computational physics courses. But object oriented programming is usually avoided in the high performance part of a computation. So, one usually uses those languages in a procedural style of programming, cheating herself that she is actually learning the advantages of object oriented programming.

is to let Fortran do what is best for, number crunching, and leave data manipulation and file administration to external, powerful tools. Tools, like `awk`, shell scripting, `gnuplot`, Perl and others, are quite powerful and complement all the weaknesses of Fortran mentioned before. The plotting program is chosen to be `gnuplot`, which provides very powerful tools to manipulate the data and create massive and complicated plots. It can also create publication quality plots and contribute to the "fun part" of the learning experience by creating animations, interactive 3d plots etc. All the tools used in the book are open source software and they are accessible to everyone for free. They can be used in a Linux environment, but they can also be installed and used in Microsoft Windows and Mac OS X.

The other hard part in teaching computational physics to scientists and engineers is to explain that the approach of solving a problem numerically is quite different from solving it analytically. Usually, students of this level are coming with a background in analysis and fundamental physics. It is hard to put them into the mode of thinking about solving a problem using only additions, multiplications and some logical operations. The hardest part is to explain the discretization of a model defined analytically, which can be done in many ways, depending on the accuracy of the approximation. Then, one has to extrapolate the numerical solution, in order to obtain a good approximation of the analytic one. This is done step by step in the book, starting with problems in simple motion and ending with discussing finite size scaling in statistical physics models in the vicinity of a continuous phase transition.

The book comes together with additional material which can be found at the web page of the book[11]. The accompanying software contains all the computer programs presented in the book, together with useful tools and programs solving some of the exercises of each chapter. Each chapter has problems complementing the material covered in the text. The student needs to solve them in order to obtain hands on experience in scientific computing. I hope that I have already stressed enough that, in order for this book to be useful, it is not enough to be read in a café or in a living room, but one needs to *do* what it says.

Hoping that this book will be useful to you as a student or as an instructor, I would like to ask you to take some time to send me feedback for improving and/or correcting it. I would also appreciate fan mail or, if you are an expert, a review of the book. If you use the book in a class, as a main textbook or as supplementary material, I would also be

[11] `www.physics.ntua.gr/~konstant/ComputationalPhysics/`

thrilled to know about it. Send me email at `konstantmail.ntua.gr` and let me know if I can publish, anonymously or not, (part of) what you say on the web page (otherwise I will only use it privately for my personal ego-boost). Well, nothing is given for free: As one of my friends says, some people are payed in dollars and some others in ego-dollars!

Have fun computing scientifically!

Athens, 2014.

# Chapter 1

# The Computer

The aim of this chapter is to lay the grounds for the development of the computational skills which are necessary in the following chapters. It is not an in depth exposition but a practical training by example. For a more systematic study of the topics discussed, we refer to the bibliography. Many of the references are freely available in the web.

The are many choices that one has to make when designing a computer project. These depend on the needs for numerical efficiency, on available programming hours, on the needs for extensibility and upgradability and so on. In this book we will get the flavor of a project that is mostly scientifically and number crunching oriented. One has to make the best of the available computing resources and have powerful tools available for a productive analysis of the data. Such an environment, found in most of today's supercomputers, that offers flexibility, dependability, simplicity, powerful tools for data analysis and effective compilers is provided by the family of the Unix operating systems. The GNU/Linux operating system is a Unix variant that is freely available and most of its utilities are open source software. The voluntary work of millions of excellent programmers worldwide has built the most stable, fastest and highest quality software available for scientific computing today. Thanks to the idea of the open source software pioneered by Richard Stallman[1] this giant collaboration has been made possible.

Another choice that we have to make is the programming language, and this is going to be Fortran. Fortran has been built mainly for numerical applications and it has been used by many scientists and engineers because of its efficiency in high performance computing. The language is simple and compilers are able to optimize, parallelize and vectorize the

[1] `www.stallman.org`

code very efficiently. There is a lot of scientific and engineering software available in libraries written in Fortran, which has been used and tested extensively for many years. This is a crucial factor for scientific software, so that it can be trusted to be efficient and free of errors. Fortran is not the best choice for interacting with the operating system or for text processing. This shortcoming can be easily overcome by the use of external tools and Fortran can be left to do what she has been designed for: number crunching. Its structure is simple and can be used both for procedural and object oriented programming, in such a way that, it will not make the life of an inexperienced programmer difficult, and at the same time provide high level, abstract and powerful tools for high performance, modular, object oriented, programming needed in a large and complicated project.

Fortran, as well as other languages like C, C++ and Java, is a language that needs to be *compiled* by a compiler. Other languages, like perl, awk, shell scripting, Macsyma, Mathematica, Octave, Matlab, ..., are *interpreted* line by line. These languages can be simple in their use, but they can be prohibitively slow when it comes to a numerically demanding program. A compiler is a tool that analyzes *the whole program* and optimizes the computer instructions executed by the computer. But if programming time is more valuable, then a simple, interpreted language can lead to faster results.

Another choice that we make in this book, and we mention it because it is not the default in most Linux distributions, is the choice of shell. The shell is a program that "connects" the user to the operating system. In this book, we will teach how to use a shell[2] to "send" commands to the operating system, which is the most effective way to perform complicated tasks. We will use the shell `tcsh`, although most of the commands can be interpreted by most popular shells. Shell scripting is simpler in this shell, although shells like `bash` provide more powerful tools, mostly needed for complicated system administration tasks. That may cause a small inconvenience to some readers, since `tcsh` is not preinstalled in Linux distributions[3].

[2]It is more popular to be called "the command line", or the "terminal", or the "console", but in fact the user interaction is through a shell.

[3]See www.tcsh.org. On Debian like systems, like Ubuntu, installation is very simple through the software center or by the command `sudo apt-get install tcsh`.

## 1.1 The Operating System

The Unix family of operating systems offer an environment where complicated tasks can be accomplished by *combining* many different tools, each of which performs a distinct task. This way, one can use the power of each tool, so that trivial but complicated parts of a calculation don't have to be programmed. This makes the life of a researcher much easier and much more productive, since research requires from us to try many things before we understand how to compute what we are looking for.

In the Unix operating system everything is a *file*, and files are organized in a unique and unified *filesystem*. Documents, pictures, music, movies, executable programs are files. But also directories or devices, like hard disks, monitors, mice, sound cards etc, are, from the point of view of the operating system, files. In order for a music file to be played by your computer, the music data needs to be written to a device file, connected by the operating system to the sound card. The characters you type in a terminal are read from a file "the keyboard", and written to a file "the monitor" in order to be displayed. Therefore, the first thing that we need to understand is the structure of the Unix filesystem.

### 1.1.1 Filesystem

There is at least one *path* in the filesystem associated with each file. There are two types of paths, *relative paths* and *absolute paths*. These are two examples:

```
bin/RungeKutta/rk.exe
/home/george/bin/RungeKutta/rk.exe
```

The paths shown above may refer to the same or a different file. This depends on "where we are". If "we are" in the directory `/home/george`, then both paths refer to the same file. If on the other way "we are" in a directory `/home/john` or `/home/george/CompPhys`, then the paths refer[4] to two different files. In the last two cases, the paths refer to the files

```
/home/john/bin/RungeKutta/rk.exe
/home/george/CompPhys/bin/RungeKutta/rk.exe
```

[4]Some times two or more paths refer to the same file, or as we say, a file has two or more "links" in the same filesystem, but let's keep it simple for the moment.

respectively. How can we tell the difference? An absolute path always begins with the / character, whereas a relative path does not. When we say that "we are in a directory", we refer to a position in the filesystem called the *current directory*, or *working directory*. Every process in the operating system has a unique current directory associated with it.

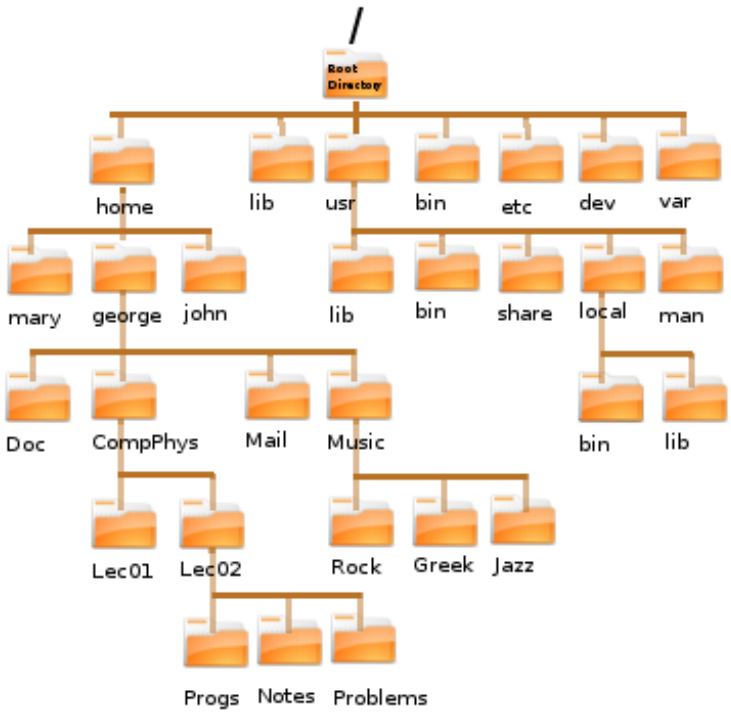



Figure 1.1: The Unix filesystem. It looks like a tree, with the *root directory* / at the top and branches that connect directories with their parents. Every directory contains files, among them other directories called its *subdirectories*. Every directory has a unique *parent directory*, noted by .. (double dots). The parent of the root directory is itself.

The filesystem is built on its *root* and looks like a tree positioned upside down. The symbol of the root is the character / The root is a *directory*. Every directory is a file that contains a list of files, and it is connected to a unique directory, its *parent directory* . Its list of files contains other directories, called its *subdirectories*, which all have it as their parent directory. All these files are the *contents* of the directory. Therefore, the filesystem is a tree of directories with the root directory at its top which branch to its subdirectories, which in their turn branch into other subdirectories and so on. There is practically no limit to how

large this tree can become, even for a quite demanding environment[5].

A path consists of a string of characters, with the characters / separating its *components*, and refers to a unique location in the filesystem. Every component refers to a file. All, but the last one, must be directories in a hierarchy, from parent directory to subdirectory. The only exception is a possible / in the beginning, which refers to the root directory. Such an example can be seen in figure 1.1.

In a Unix filesystem there is complete freedom in the choice of the location of the files[6]. Fortunately, there are some universally accepted conventions respected by almost everyone. One expects to find *home directories* in the directory `/home`, configuration files in the directory `/etc`, application executables in directories with names such as `/bin`, `/usr/bin`, `/usr/local/bin`, software libraries in directories with names such as `/lib`, `/usr/lib` etc.

There are some important conventions in the naming of the paths. A single dot "`.`" refers to the current directory and a double dot "`..`" to the parent directory. Similarly, a tilde "~" refers to the home directory of the user. Assume, e.g., that we are the user `george` running a process with a current directory `/home/george/Music/Rock` (see figure 1.1). Then, the following paths refer to the same file `/home/george/Doc/lyrics.doc`:

```
../../Doc/lyrics.doc
~/Doc/lyrics.doc
~george/Doc/lyrics.doc
./../../Doc/lyrics.doc
```

Notice that ~ and `~george` refer to the home directory of the user `george` (ourselves), whereas `~mary` refer to the home directory of another user, `mary`.

We are now going to introduce the basic commands for filesystem navigation and manipulation[7]. The command `cd` (=change directory)

[5]Of course, the capacity of the filesystem is finite, issue the command "`df -i .`" in order to see the number of inodes available in your filesystem. Every file corresponds to one and only one inode of the filesystem. Every path is mapped to a unique inode, but an inode maybe pointed to by more than one paths.

[6]This gives a great sense of freedom, but historically this was a important factor that led the Unix operating systems, although superior in quality, not to win a fair share of the market! The Linux family tries to keep things simple and universal to a large extent, but one should be aware that because of this freedom files in different version of Linuxes or Unices can be in different places.

[7]Remember that lines that begin with the > character are commands. All other lines refer to the output of the commands.

changes the current directory, whereas the command `pwd` (=print working directory) prints the current directory:

```
> cd /usr/bin
> pwd
/usr/bin
> cd /usr/local/lib
> pwd
/usr/local/lib
> cd
> pwd
/home/george
> cd -
> pwd
/usr/local/lib
> cd ../../
> pwd
/usr
```

The argument of the command `cd` is an absolute or a relative path. If the path is correct and we have the necessary permissions, the command changes the current directory to this path. If no path is given, then the current directory changes to the home directory of the user. If the character `-` is given instead of a path, then the command changes the current directory to the previous current directory.

The command `mkdir` creates new directories, whereas the command `rmdir` removes empty directories. Try:

```
> mkdir new
> mkdir new/01
> mkdir new/01/02/03
mkdir: cannot create directory 'new/01/02/03': No such file or
       directory
> mkdir -p new/01/02/03
> rmdir new
rmdir: 'new': Directory not empty
> rmdir new/01/02/03
> rmdir new/01/02
> rmdir new/01
> rmdir new
```

Note that the command `mkdir` cannot create directories more than one level down the filesystem, whereas the command `mkdir -p` can. The "switch" `-p` makes the behavior of the command different than the default one.

In order to list the contents of a directory, we use the command `ls` (=list):

```
> ls
BE.eps   Byz.eps   Programs       srBE_xyz.eps   srB_xyz.eps
B.eps    Bzy.eps   srBd_xyz.eps   srB_xy.eps
> ls Programs
Backup              rk3_Byz.f90   rk3.f90
plot-commands       rk3_Bz.f90    rk3_g.f90
```

The first command is given without an argument and it lists the contents of the current directory. The second one, lists the contents of the subdirectory of the current directory `Programs`. If the argument is a list of paths pointing to regular files, then the command prints the names of the paths. Another way of giving the command is

```
total 252
-rw-r--r--  1 george users 24284 May  1 12:08 BE.eps
-rw-r--r--  1 george users 22024 May  1 11:53 B.eps
-rw-r--r--  1 george users 29935 May  1 13:02 Byz.eps
-rw-r--r--  1 george users 48708 May  1 12:41 Bzy.eps
drwxr-xr-x  4 george users  4096 May  1 23:38 Programs
-rw-r--r--  1 george users 41224 May  1 22:56 srBd_xyz.eps
-rw-r--r--  1 george users 23187 May  1 21:13 srBE_xyz.eps
-rw-r--r--  1 george users 24610 May  1 20:29 srB_xy.eps
-rw-r--r--  1 george users 23763 May  1 20:29 srB_xyz.eps
```

The switch `-l` makes `ls` to list the contents of the current directory together with useful information on the files in 9 columns. The first column lists the *permissions* of the files (see below). The second one lists the number of links of the files[8]. The third one lists the user who is the *owner* of each file. The fourth one lists the group that is assigned to the files. The fifth one lists the size of the file in bytes (=8 bits). The next three ones list the modification time of the file and the last one the paths of the files.

File permissions[9] are separated in three classes: owner permissions, group permissions and other permissions. Each class is given three specific permissions, `r`=**r**ead, `w`=**w**rite and `x`=e**x**ecute. For regular files, read permission effectively means access to the file for reading/copying, write permission means permission to modify the contents of the file and ex-

[8]For a directory it means the number of its subdirectories plus 2 (the parent directory and itself). For a regular file, it shows how many paths in the filesystem point to this file.

[9]See the "File system permissions" entry in `en.wikipedia.org`.

ecute permission means permission to execute the file as a command[10]. For directories, read permission means that one is able to read the names of the files in the directory (but not make it as current directory with the `cd` command), write permission means to be able to modify its contents (i.e. create, delete, and rename files) and execute permission grants permission to access/modify the contents of the files (but not list the names of the files, this is granted by the read permission).

The command `ls -l` lists permissions in three groups. The owner (positions 2-4), the group (positions 5-7) and the rest of the world (others - positions 8-10). For example

```
-rw-r--r--
-rwxr-----
drwx--x--x
```

In the first case, the owner has read and write but not execute permissions and the group+others have only read permissions. In the second case, the user has read, write and execute permissions, the group has read permissions and others have no permissions at all. In the last case, the user has read, write and execute permissions, whereas the group and the world have only execute permissions. The first character `d` indicates a special file, which in this case is a **d**irectory. All special files have this position set to a character, while regular files have it set to `-`.

File permissions can be modified by using the command `chmod`:

```
> chmod u+x  file
> chmod og-w file1 file2
> chmod a+r  file
```

Using the first command, the owner (`u`≡ user) obtains (`+`) permission to execute (`x`) the file named `file`. Using the second one, the rest of the world (`o`≡ others) and the group (`g`≡group) loose (`-`) the write (`w`) permission to the files named `file1` and `file2`. Using the third one, everyone (`a`≡all) obtain read (`r`) permission on the file named `file`.

We will close this section by discussing some commands which are used for administering files in the filesystem. The command `cp` (copy) copies the contents of files into other files:

```
> cp file1.f90 file2.f90
```

[10]Of course it is the user's responsibility to make sure the file with execute permission is actually a program that is possible to execute. An error results if this is not the case.

```
> cp file1.f90 file2.f90 file3.f90 Programs
```

If the file `file2.f90` does not exist, the first command copies the contents of `file1.f90` to a new file `file2.f90`. If it already exists, it replaces its contents by the contents of the file `file2.f90`. In order for the second command to be executed, `Programs` needs to be a directory. Then, the contents of the files `file1.f90, file2.f90, file3.f90` are copied to indentical files in the directory `Programs`. Of course, we assume that the user has the appropriate privileges for the command to be executed successfully.

The command `mv` "moves", or renames, files:

```
> mv file1.f90 file2.f90
> mv file1.f90 file2.f90 file3.f90 Programs
```

The first command renames the file `file1.f90` to `file2.f90`. The second one moves files `file1.f90, file2.f90, file3.f90` into the directory `Programs`.

The command `rm` (remove) deletes files[11]. Beware, the command is unforgiving: after deletion, a file cannot be restored into the filesystem[12]. Therefore, after executing successfully the following commands

```
> ls
file1.f90  file2.f90  file3.f90  file4.csh
> rm file1.f90 file2.f90 file3.f90
> ls
file4.csh
```

the files `file1.f90, file2.f90, file3.f90` do not exist in the filesystem anymore. A more prudent use of the command demands the flag `-i`. Then, before deletion we are asked for confirmation:

```
> rm -i *
rm: remove regular file 'file1.f90'? y
rm: remove regular file 'file2.f90'? y
rm: remove regular file 'file3.f90'? y
rm: remove regular file 'file4.csh'? n
```

[11]Actually it removes "links" from files. A file may have more than one links in the same partition of a filesystem. A file is deleted when its last link is removed.

[12]This does not mean that its contents have been deleted from the disk. Deletion means marking for overwriting. Until the data is overwritten it can be recovered by the use of special tools. Shredding sensitive data can be tricky business...

```
> ls
file4.csh
```

When we type y, the file is deleted, when we type n, the file is not deleted.

We cannot remove directories the same way. It is possible to use the command rmdir in order to remove *empty* directories. In order to delete directories together with their contents (including subdirectories and their contents) use the command[13] rm -r. For example, assume that the contents of the directories dir1 and dir1/dir2 are the files:

```
./dir1
./dir1/file2.f90
./dir1/file1.f90
./dir1/dir2
./dir1/dir2/file3.f90
```

Then the results of the following commands are:

```
> rm dir1
rm: cannot remove ‘dir1’: Is a directory
> rm dir1/dir2
rm: cannot remove ‘dir1/dir2’: Is a directory
> rmdir dir1
rmdir: dir1: Directory not empty
> rmdir dir1/dir2
rmdir: dir1/dir2: Directory not empty
> rm -r dir1
```

The last command removes all files (assuming that we have write permissions for all directories and subdirectories). Alternatively, we can empty the contents of all directories first, and then remove them with the command rmdir:

```
> cd dir1/dir2; rm file3.f90
> cd .. ; rmdir dir2
> rm file1.f90 file2.f90
> cd .. ; rmdir dir1
```

Note that by using a semicolon, we can execute two or more commands on the same line.

[13] A small mistake, like rm -rf * and your data is ... history!

### 1.1.2 Commands

Commands in a Unix operating system are files with execute permission. When we write a sentence on the command line, like

```
> ls -l test.f90 test.dat
```

the shell reads its and interprets it. The shell is a program that creates a interface between a user and the operating system. The first word (`ls`) of the sentence is interpreted as a command. The rest of the words are the *arguments* of the command and the program can use them (or not) at the discretion of its programmer. There is a special convention for arguments that begin with a `-` (e.g. `-l`, `--help`, `--version`, `-O3`). They are called *options* or *switches*, and they act as virtual switches that make the program act in a particular way. We have already seen that the program `ls` gives a different output with the switch `-l`.

In order for a command to be executed, the shell looks for a file that has the same name as the command (here a file named `ls`). In order to understand where the shell looks for such a file, we should digress a little bit and explain the use of *shell variables* and *environment variables*. These have a name, which is a string of permissible characters, and their values are obtained by preceding their name with the `$` character. For example the variable `PATH` has value `$PATH`. The values of the environment variables can be set with the command[14] `setenv` and of the shell variables with the command `set`:

```
> setenv MYVAR test-env
> set myvar = test-shell
> echo $MYVAR $myvar
test-env test-shell
```

Two special variables are the variables `PATH` and `path`:

```
>echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/X11/bin
>echo $path
/usr/local/bin /usr/bin /bin /usr/X11/bin
```

The first one is an environment variable and the second one is a shell variable. Their values are set by the shell, and we don't need to worry

[14] The command `setenv` is special to the `tcsh` shell. For example the `bash` shell uses the syntax `MYVAR=test-env` in order to set the value of an environment variable.

about them, unless we want to change them. Their value is a string of characters whose components should be valid paths to directories. In the first case, the components are separated by a :, while in the second case, by one or more spaces. In the example shown above, the shell searches each component of the `path` or `PATH` variables (in this order) until it finds a file `ls` in their contents. If it succeeds and the file has execute permissions, then the program in this file is executed. If it fails, then it prints an error message. Try the commands:

```
> which ls
/bin/ls
> ls -l /bin/ls
-rwxr-xr-x 1 root root 93560 Sep 28  2006 /bin/ls
```

We see that the program that the `ls` command executes the program in the file `/bin/ls`.

The arguments of a command are passed on to the program that the command executes for possible interpretation. For example:

```
> ls -l test.f90 test.dat
```

The argument `-l` is the switch that results in a long listing of the files. The arguments `test.f90` and `test.dat` are interpreted by the program `ls` as paths that it will look up for file information.

You can use the * (wildcard) character as a shorthand notation for a group of files. For example, in the command shown below

```
> ls -l *.f90 *.dat
```

the shell will expand `*.f90` and `*.dat` to a list of all files whose names and with `.f90` or `.dat`. Therefore, if the current directory contains the files `test.f90`, `test1.f90`, `myprog.f90`, `test.dat`, `hello.dat`, the arguments that will be passed on to the command `ls` are

```
> ls -l myprog.f90 test1.f90 test.f90 hello.dat test.dat
```

For each command there are three special files associated with it. The first one is the *standard input* (`stdin`), the second one is the *standard output* (`stdout`) and the third one the *standard error* (`stderr`). These are files where the program can print or read data from. By default, these files are the terminal that the user uses to execute the command. In this case,

when the program reads data from the stdin, then it reads the data that we type to the terminal using the keyboard. When the program writes data to the stdout or to the stderr, then the data is written to the terminal.

The advantage of using these special files in order to read/write data is that the user can *redirect* the input/output to these files to any file she wants. Using the character > at the end of a command redirects the stdout to the file whose name is written after >. For example:

```
> ls
file1.f90  file2.f90  file3.f90  file4.csh
> ls > results
> ls
file1.f90  file2.f90  file3.f90  file4.csh  results
```

The first of the above commands, prints the contents of the current working directory to the terminal. The second command redirects data written to the stdout to the file results. After executing the command, the file results is created and its contents are the names of the files file1.f90 file2.f90 file3.f90 file4.csh. If the file results does not exist (as in the above example), the file is created. If it already exists, it is *truncated* and its contents replaced by the data written to the stdout of the command. If we want to *append* data without erasing the existing contents, then we should use the string of characters >>. Therefore, if we give the command

```
> ls >> results
```

after executing the previous commands, then the contents of the file results will be

```
file1.f90  file2.f90  file3.f90  file4.csh
file1.f90  file2.f90  file3.f90  file4.csh  results
```

The redirection of the stdin is accomplished by the use of the character < while that of the stderr by the use of the string of characters[15] >&. We will see more examples in section 1.2.

It is possible to redirect the stdout of a command to be the stdin of another command. This is very useful for creating *filters*. A filter is

[15]This syntax is particular to the tcsh shell. For other shells (bash, sh, ...) read their documentation.

a command that creates a flow of data between two or more programs. This process is called *piping*. Pipes are creating by using the character |

```
> cmd1 | cmd2 | cmd3 | ... | cmdN
```

Using the syntax shown above, the `stdout` of the command `cmd1` is redirected to the `stdin` of the command `cmd2`, the `stdout` of the command `cmd2` is redirected to the `stdin` of the command `cmd3` etc. More examples will be presented in section 1.2.

### 1.1.3 Looking for Help

Unix got itself a reputation for not being user friendly. This is far from the truth. Although there is a steep learning curve, detailed documentation for almost everything is available online.

The key for a comfortable ride is to learn how to use the help system available on your computer and on the internet. Most of the commands are self documented. A simple test, like the one shown below, will help you with the basic usage of most of the commands:

```
> cmd --help
> cmd -h
> cmd -help
> cmd -\?
```

For example, try the command `ls --help`. For a window application, start from the menu "Help". You should not be afraid and/or lazy and you should proceed with careful searching and reading.

For example, let's assume that you have heard about a command that sounds like `printf`, or something like that. The first level of online help is the `man` (=manual) command that searches the "man pages". Read the output of the command

```
> man printf
```

The command `info` usually provides more detailed and user friendly documentation. It has basic browsing capabilities like the browsers you use to read pages from the internet. Try the command

```
> info printf
```

Furthermore, the commands

```
> man -k printf
> whatis printf
```

will inform you that there are other, possibly related, commands with names like `fprintf`, `fwprintf`, `wprintf`, `sprintf`...:

```
> whatis printf
printf                (1)   - format and print data
printf                (1p)  - write formatted output
printf                (3)   - formatted output conversion
printf                (3p)  - print formatted output
printf [builtins]     (1)   - bash built-in commands, see bash↩
   (1)
```

The second column printed by the `whatis` command is the "section" of the man pages. In order to gain access to the information in a particular section, you have to give it as an argument to the `man` command:

```
> man 1  printf
> man 1p printf
> man 3  printf
> man 3p printf
> man bash
```

Section 1 of the man pages contains information of ordinary command line commands, section 3 contains information on functions in libraries of the C language. Section 2 contains information on commands used for system administration. You may browse the directory `/usr/share/man`, or read the man page of the `man` command (use the command `man man` for that!).

By using the command

```
> printf --help
```

we obtain plenty of memory refreshing information. The command

```
> locate printf
```

shows us many files related to the command `printf`. The commands

```
> which printf
> where printf
```

give information on the location of the executable(s) of the command `printf`.

Another useful feature of the shell is the *command* or it filename completion. This means that we can write only the first characters of the name of a command or filename and then press simultaneously the keys `[Ctrl-d]`[16] (i.e. press the key `Ctrl` and the key of the letter `d` at the same time). Then the shell will complete the name of the command up to the point that is is unique with the given string of characters[17]:

```
> pri[Ctrl-d]
printafm    printf  printenv   printnodetest
```

Try to type an `x` on the command line and then type `[Ctrl-d]`. You will learn all the commands that are available and whose name begins with an `x`: `xterm, xeyes, xclock, xcalc, ...`

Finally, the internet contains a wealth of information. Google your blues... and you will be rewarded!

## 1.2 Text Processing Tools – Filters

For doing data analysis, we will need powerful tools for manipulating data in text files. These are files that consist solely of printable characters. Some tools that can be used in order to construct complicated and powerful filters are the programs `cat, less, head, tail, grep, sort` and `awk`.

Suppose that we have data in a file named `data`[18] which contains information on the contents of a food warehouse and their prices:

```
bananas  100 pieces 1.45
apples   325 boxes  1.18
pears     34 kilos  2.46
bread     62 kilos  0.60
```

[16] If you use the `bash` shell press `[Tab]` once or twice.

[17] Use the same procedure to auto-complete the names of files in the arguments of commands.

[18] The particular file, as well as most of the files in this section, can be found in the accompanying software of the chapter. It is highly recommended that you try all the commands in this section by using all the provided files.

```
ham        85 kilos   3.56
```

The command

```
> cat data
```

prints the contents of the file `data` to the `stdout`. In general, this command prints the contents of all files given in its arguments or the `stdin` if none is given. Since the `stdin` and the `stdout` can be redirected, the command

```
> cat <  data > data1
```

takes the contents of the file `data` from the `stdin` and prints them to the `stdout`, which in this case is the file `data1`. This command has the same result as the command:

```
> cp data data1
```

The command

```
> cat data data1 > data2
```

prints the contents of the file `data` and then the contents of the file `data1` to the `stdout`. Since the `stdout` is redirected to the file `data2`, `data2` contains the data of both files.

By giving the command

```
> less gfortran.txt
```

you can browse the data contained in the file `gfortran.txt` one page at a time. Press [space] in order to "turn" a page, [b] to turn back a page. Press the up and down arrows to move one line backwards/forward. Press [g] in order to jump to the beginning of the file and press [G] in order to jump to the end. Press [h] in order to get a help message and press [q] in order to quit.

The commands

```
> head -n 1 data
bananas  100 pieces 1.45
> tail -n 2 data
```

```
bread      62 kilos  0.60
ham        85 kilos  3.56
> tail -n 2 data | head -n 1
bread      62 kilos  0.60
```

print the first line, the last two lines and the second to the last line of the file `data` to the `stdout` respectively. Note that, by piping the `stdout` of the command `tail` to the `stdin` of the command `head`, we are able to construct the filter "print the line before the last one".

The command `sort` sorts the contents of a file by comparing each line of its text with all others. The sorting is alphabetical, unless otherwise set by using options. For example

```
> sort data
apples    325 boxes  1.18
bananas   100 pieces 1.45
bread      62 kilos  0.60
ham        85 kilos  3.56
pears      34 kilos  2.46
```

For reverse sorting, try `sort -r data`. We can also sort by comparing specific *fields* of each line. By default, fields are words separated by one or more spaces. For example, in order to sort w.r.t. the second column of the file `data`, we can use the switch `-k 2` (=second field). Furthermore, we can use the switch `-n` for numerical sorting:

```
> sort -k 2 -n data
pears      34 kilos  2.46
bread      62 kilos  0.60
ham        85 kilos  3.56
bananas   100 pieces 1.45
apples    325 boxes  1.18
```

If we omit the switch `-n`, the comparison of the lines is performed based on character sorting of the second field and the result is

```
> sort -k 2 data
bananas   100 pieces 1.45
apples    325 boxes  1.18
pears      34 kilos  2.46
bread      62 kilos  0.60
ham        85 kilos  3.56
```

The last column contains floating point numbers (not integers). In order to sort by the values of such numbers we should use the switch `-g`:

```
> sort -k 4 -g data
bread     62 kilos  0.60
apples   325 boxes  1.18
bananas  100 pieces 1.45
pears     34 kilos  2.46
ham       85 kilos  3.56
```

The command `grep` processes a text file line by line, searching for a given string of characters. When this string is found anywhere in a line, this line is printed to the `stdout`. The command

```
> grep kilos data
pears     34 kilos  2.46
bread     62 kilos  0.60
ham       85 kilos  3.56
```

prints each line containing the string "kilos". If we want to search for all line *not* containing the string "kilos", then we add the switch `-v`:

```
> grep -v kilos data
bananas  100 pieces 1.45
apples   325 boxes  1.18
```

We can use a *regular expression* for searching a whole family of strings of characters. These monsters need a full book for discussing them in detail! But it is not hard to learn how to use some simple forms of regular expressions. Here are some examples:

```
> grep ^b data
bananas  100 pieces 1.45
bread     62 kilos  0.60
> grep '0$' data
bread     62 kilos  0.60
> grep '3[24]' data
apples   325 boxes  1.18
pears     34 kilos  2.46
```

The first one, prints each line whose *first* character is a `b`. The second one, prints each line that *ends* with a `0`. The third one, prints each line contaning the strings `32` *or* `34`.

By far, the strongest tool in our toolbox is the `awk` program. By default, `awk` analyzes a text file line by line. Each word (or *field* in the awk jargon) of these lines is stored in a set of variables with names `$1`, `$2`, .... The variable `$0` contains the full line currently processed, whereas the variable `NF` counts the number of fields in the current line. The variable `NR` counts the number of lines of the file processed so far by `awk`.

An `awk` program can be written in the command line. A set of commands within `{ ... }` is executed for each line of input. The constructs `BEGIN{ ... }` and `END{ ... }` contain commands executed, only once, before and after the processing of the file respectively. For example, the command

```
> awk '{print $1,"total value= ",$2*$4}' data
bananas total value=  145
apples  total value=  383.5
pears   total value=  83.64
bread   total value=  37.2
ham     total value=  302.6
```

prints the name of the product (1st column = `$1`) and the total value stored in the warehouse (2nd column = `$2`) $\times$ (4th column = `$4`). More examples are given below:

```
> awk '{value += $2*$4}END{print "Total= ",value}' data
Total=  951.94
> awk '{av += $4}END{print "Average Price= ",av/NR}' data
Average Price=  1.85
> awk '{print $2^2 * sin($4) + exp($4)}' data
```

The first one calculates the total value of all products: The processing of each line results in the increment (`+=`) of the variable `value` by the product of the second and fourth fields. In the end (`END{ ... }`), the string `Total=` is printed, together with the final value of the variable `value`. This is an easy way for computing the sum of the values calculated for each line. The second command, calculates and prints an average. The sum is calculated in each line and stored in the variable `av`. In the end, we print the quotient of the sum of all values by the number of lines that have been processed (`NR`). The last command shows a (crazy) mathematical expression based on numerical values found in each line of the file `data`: It computes the square of the second field times the sine of the fourth field plus the exponential of the fourth field.

There is much more potential in the commands presented above. Reading the documentation and getting experience by using them will provide you with very strong tools in order to accomplish complicated tasks.

## 1.3 Programming with Emacs

For a programmer that spends many hours programming every day, the environment and the tools available for editing the commands of a large and complicated program determine, to a large extent, the quality of her life! An editor edits the contents of a text file, that consists solely of printable characters. Such editors, available in most Linux environments, are the programs `gedit, vim, pico, nano, zile...` They provide basic functionality such as adding, removing or changing text within a file as well as more complicated functions, such as copying, pasting, searching and replacing text etc. There are many functions that are particularly useful to a programmer, such as detecting and formatting keywords of a particular programming language, pretty printing, closing scopes etc, which can be very useful for comfortable programming and for spotting errors. A very powerful and "knowledgeable" editor, offering many such functions for several programming languages, is the GNU Emacs editor[19]. Emacs is open source software, it is available for free and can be used in most available operating systems. It is programmable[20] and the user can automate most of her everyday repeated tasks and configure it to her liking. There is a full interaction with the operating system, in fact Emacs has been built with the ambition of becoming an operating system. For example, a programmer can edit a Fortran file, compile it, debug it and run it, everything done with Emacs commands.

### 1.3.1 Calling Emacs

In the command line type

```
> emacs &
```

[19] `http://www.gnu.org/software/emacs/` (main site), `http://www.emacswiki.org/` (expert tips), `http://en.wikipedia.org/wiki/Emacs` (general info)

[20] Emacs is written in a dialect of the programming language Lisp, called Elisp. There is no need of an in-depth knowledge of the language in order to program simple functions, just see how others are doing it...

Note the character & at the end of the line. This makes the particular command to run in the *background*. Without it, the shell waits until a command exits in order to return the prompt.

In a desktop environment, Emacs starts in its own window. For a quick and dirty editing session, or in the case that a windows environment is not available[21], we can run Emacs in a terminal mode. Then, we omit the & at the end of the line and we run the command

```
> emacs -nw
```

The switch `-nw` forces Emacs to run in terminal mode.

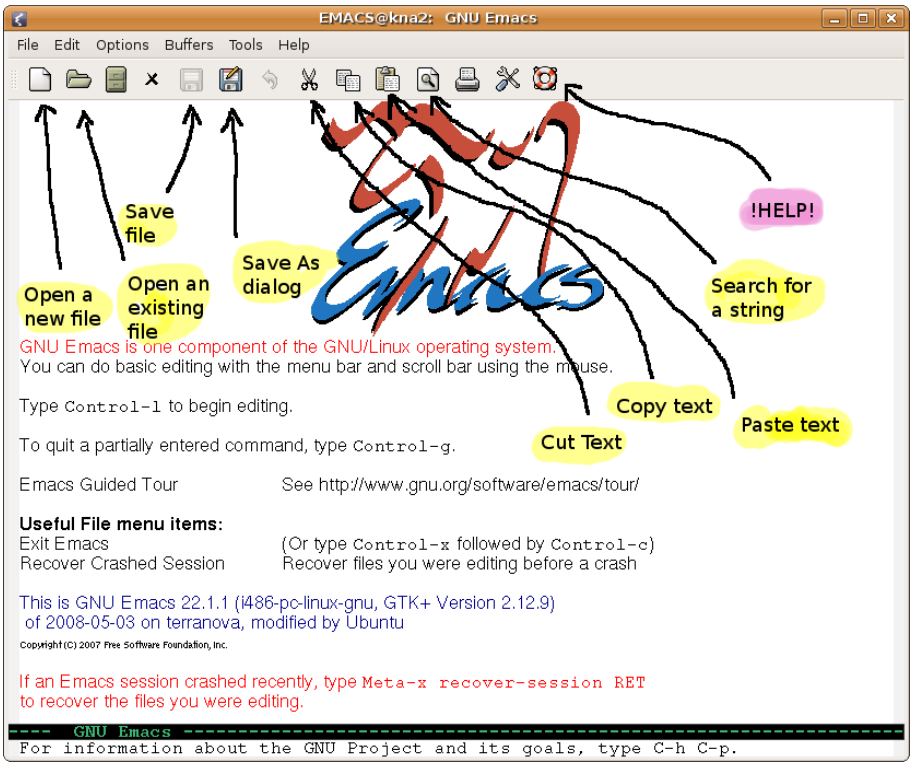



Figure 1.2: The Emacs window in a windows environment. The buttons of very basic functions found on its toolbar are shown and explained.

[21] Quite handy when we edit files in a remote computer.

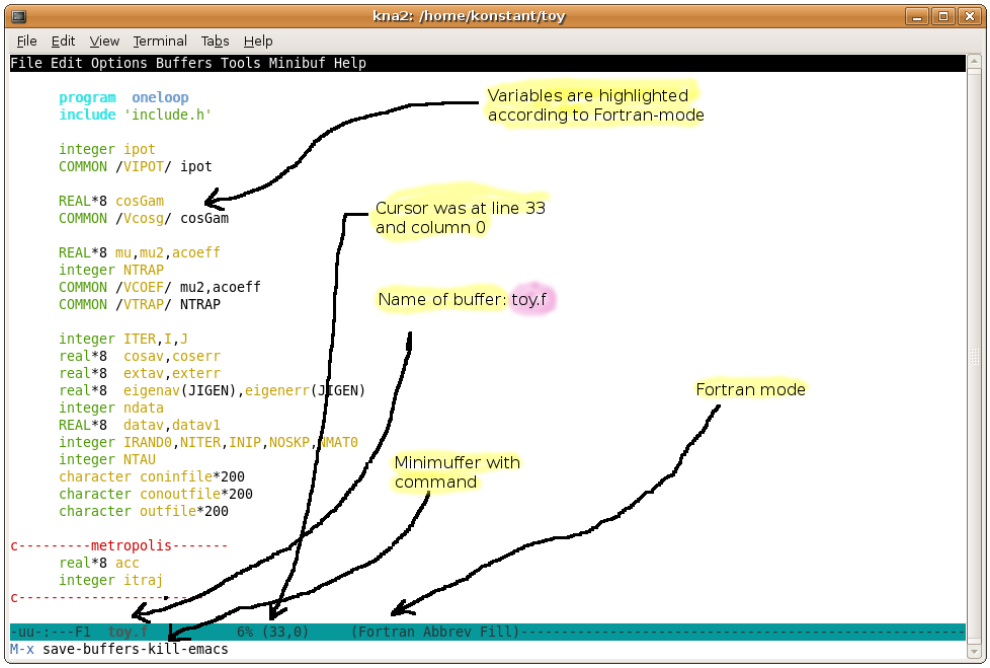



Figure 1.3: Emacs in a non-window mode running on the console. In this figure, we have typed the command `save-buffers-kill-emacs` in the minibuffer, a command that exits Emacs after saving edited data from all buffers. The same command can be given using the keyboard shortcut `C-x C-c`. We can see the *mode line* and the name of the buffer `toy.f` written on it, the percentage of the buffer (6%) shown in the window, the line and columns (33,0) where the point lies and the editing *mode* which is active on the buffer (Fortran mode (Fortran), Abbreviation mode (Abbrev), Auto Fill mode (Fill)).

### 1.3.2 Interacting with Emacs

We can interact with Emacs in various ways. Newbies will prefer buttons and menus that offer a simple and intuitive interface. For advanced usage, however, we recommend that you make an effort to learn the keyboard shortcuts. There are also thousands of functions available to be used interactively. They are called from a "command line", called *the minibuffer* in the Emacs jargon.

Keyboard shortcuts are usually combinations of keystrokes that consist of the simultaneous pressing of the `Ctrl` or `Alt` keys together with other keys. Our convention is that a key sequence starting with a `C-` means that the characters that follow are keys simultaneously pressed with the `Ctrl` key. A key sequance starting with a `M-` means that the

characters that follow are keys simultaneously pressed with the `Alt` key[22]. Some commands have shortcuts consisting of two or more composite keystrokes. For example by `C-x C-c` we mean that we have to press simultaneously the `Ctrl` key together with `x` and *then* press simultaneously the `Ctrl` key together with `c`. This sequence is a shortcut to the command that exits Emacs. Another example is `C-x 2` which means to press the `Ctrl` key together with `x` and then press only the key `2`. This is a shortcut to the command that splits a window horizontally to two equal parts.

The most useful shortcuts are `M-x` (press the `Alt` key siumutaneously with the `x` key) and `C-g`. The first command takes us to the minibuffer where we can give a command by typing its name. For example, type `M-x` and then type `save-buffers-kill-emacs` in the minibuffer (this will terminate Emacs). The second one is an "SOS button" that interrupts anything Emacs does and returns control to the working buffer. This can be pretty handy when a command hangs or destroys our work and we need to interrupt it.

The conventions for the mouse events are as follows: With `Mouse-1`, `Mouse-2` and `Mouse-3` we denote a simple click with the left, middle and right buttons of the mouse respectively. With `Drag-Mouse-1` we mean to press the left button of the mouse and at the same time drag the mouse.

We summarize the possible ways of giving a command in Emacs with the following examples that have the same effect: Open a file and put its contents in a buffer for editing.

- By pressing the toolbar button that looks like a white sheet of paper (see figure 1.2).
- By choosing the `File`→`Visit New File` menu entry.
- By typing the keyboard shortcut `C-x C-f`.
- By typing the name of the command in the minibuffer: `M-x find-file`

The number of available commands increases from the top to the bottom of the above list.

### 1.3.3 Basic Editing

In order to edit a file, Emacs places the *contents* of a file in a *buffer*. Such a buffer is a chunk of computer memory where the contents of the file

[22]Actually, `M-` is the so called `Meta` key, usually bound to the `Alt` key. It is also bound to the `Esc` and `C-[` keys. The latter can be our *only* choices available in dumb terminals.

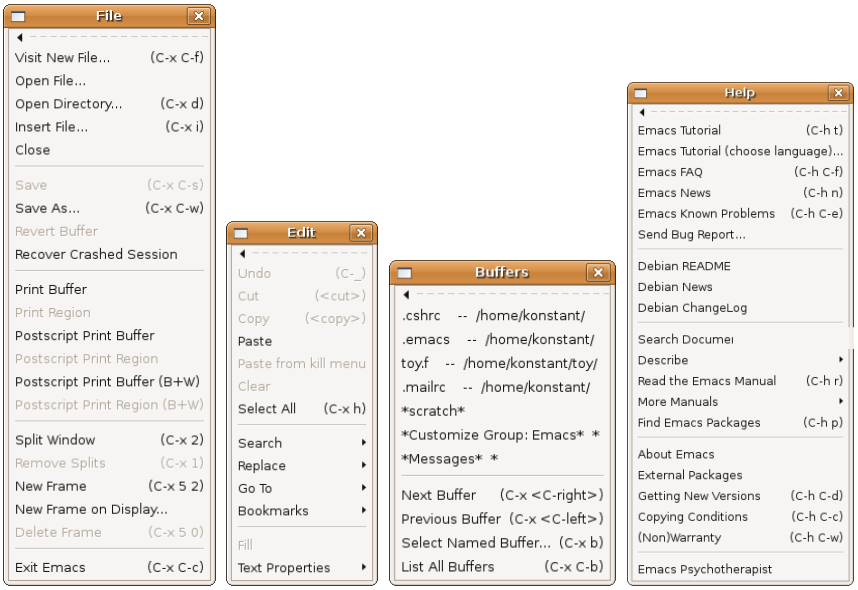



Figure 1.4: The basic menus found in Emacs when run in a desktop environment. We can see the basic commands and the keyboard shortcut reminders in the parentheses. E.g. the command `File` → `Visit New File` can be given by typing `C-x C-f`. Note the commands `File` → `Visit New File` (open a file), `File`→`Save` (write contents of a buffer to a file), `File`→`Exit Emacs`, `File` → `Split Window` (split window in two), `File`→`New Frame` (open a new Emacs desktop window) and of course the well known commands `Cut`, `Copy`, `Paste`, `Undo` from the `Edit` menu. We can choose different buffers from the menu `Buffers`, which contain the contents of other files that we have opened for editing. We recommend trying the `Emacs Tutorial` and `Read Emacs Manual` in the `Help` menu.

are copied and it is not the file itself. When we make changes to the contents of a buffer, the file remains intact. For our changes to take effect and be written to the file, we have to save the buffer. Then, the contents of the buffer are written back to the file. It is important to understand the following cycle of events:

- Read a file's contents to a buffer.
- Edit buffer contents.
- Write (save) buffer's contents back into the file.

Emacs may have more than one buffers open for editing simultaneously. By default, the name of the buffer is the same as the name of the file

that is edited, although this is not necessary[23]. The name of a buffer is written in the modeline of the window of the buffer, as can be seen in figure 1.3.

If Emacs crashes or exits before we save our edits, it is possible to recover (part of) them. There is a command `M-x recover-file` that will guide us through the necessary recovery steps, or we can look for a file that has the same name as the buffer we were editing surrounded by two `#`. For example, if we were editing the file `file.f90`, the automatically saved changes can be found in the file `#file.f90#`. Auto saving is done periodically by Emacs and its frequency can be controlled by the user.

The point where we insert text while editing is called "the point". This is right before the blinking cursor[24]. Each buffer has another position marked by "the mark". A point and the mark define a "region" in the buffer. This is a part of the text in the buffer where the functions of Emacs can act (e.g. copy, cut, change case, spelling etc.). We can set the region by setting a point and then press `C-SPC`[25] or give the command `M-x set-mark-command`. This defines the current point to be the mark. Then we can move the cursor to another point which will define a region together with the mark that we set. Alternatively we can use `Drag-Mouse-1` (hold the left mouse button and drag the mouse) and mark a region. The mark can be set with `Mouse-3`, i.e. with a simple click of the right button of the mouse. Therefore by `Mouse-1` at a point and then `Mouse-3` at a different point will set the region between the two points.

We can open a file in a buffer with the command `C-x C-f`, and then by typing its path. If the file already exists, its contents are copied to a buffer, otherwise a new buffer is created. Then:

- We can browse the buffer's contents with the Up/Down/Left/Right arrows. Alternatively, by using the commands `C-n, C-p, C-f` and `C-b`.

- If the buffer is large, we can browse its contents one page at a time

[23]The user can change the name of the buffer without affecting the name of the file it edits. Also, if we open more than one files with the same name, emacs gives each buffer a unique name. E.g. if we edit more than one files named `index.html` then the corresponding buffers are named `index.html`, `index.html<2>`, `index.html<3>`, ... .

[24]Strictly speaking, the point lies between two characters and not on top of a character. The cursor lies on the character immediately to the right of the point. A point is assigned to every *window*, therefore a buffer can have multiple points, one for each window that displays its contents.

[25]Press the `Ctrl` and spacebar keys simultanesouly.

by using the `Page Up/Page Dn` keys. Alternatively, by using the commands `C-v`, `M-v`.

- Enter text at the points simply by typing it.
- Delete characters before the point by using the `Backspace` key and after the point by using the `Delete` key. The command `C-d` deletes a forward character.
- Erase all the characters in a line that lie ahead of the point by using the command `C-k`.
- Open a new line by using `Enter` or `C-o`.
- Go to the first character of a line by using `Home` and the last one by using `End`. Alternatively, by using the commands `C-a` and `C-e`, respectively.
- Go to the first character of the buffer with the key `C-Home` and the last one with the key `C-End`. Alternatively, with `M-x beginning-of-buffer` and `M-x end-of-buffer`.
- Jump to any line we want: Type `M-x goto-line` and then the line number.
- Search for text after the point: Press `C-s` and then the text you are looking for. This is an incremental search and the point jumps immediately to the first string that matches the search. The same search can be repeated by pressing `C-s` repeatedely.

When we finish editing (or frequently enough so that we don't loose our work due to an unfortunate event), we save the changes in the buffer, either by pressing the save icon on the toolbar, or by pressing the keys `C-s`, or by giving the command `M-x save-buffer`.

### 1.3.4 Cut and Paste

Use the instructions below for slightly more advanced editing:

- `Undo`! Some of the changes described below can be catastrophic. Emacs has a great `Undo` function that keeps in its memory many of the changes inflicted by our editing commands. By repeatedely pressing `C-/`, we undo the changes we made. Alternatively, we can use `C-x u` or the menu entry `Edit`→`Undo`. Remember that `C-g` interrupts any Emacs process currently running in the buffer.

- Cut text by using the mouse: Click with `Mouse-1` at the point before the beginning of the text and then `Mouse-3` at the point after the end. A second `Mouse-3` and the region is ... gone (in fact it is written in the "kill ring" and it is available for pasting)!

- Cut text by using a keyboard shortcut: Set the mark by `C-SPC` at the point before the beginning of the text that you want to cut. Then move the cursor after the last character of the text that you want to cut and type `C-w`.

- Copy text by using the mouse: Drag the mouse `Drag-Mouse-1` and mark the region that you want to copy. Alternatively, `Mouse-1` at the point before the beginning of the text and then `Mouse-3` at the point after the end.

- Copy text by using a keyboard shortcut: Set the mark at the beginning of the text with `C-SPC` and then move the cursor after the last character of the text. Then type `M-w`.

- Pasting text with the mouse: We click the middle button[26] `Mouse-2` at the point that we want to insert the text from the kill ring (the copied text).

- Pasting text with a keyboard shortcut: We move the point to the desired insertion point and type `C-y`.

- Pasting text from previous copying: A fast choice is the menu entry `Edit`→`Paste from kill manu` and then select from the copied texts. The keyboard shortcut is to first type `C-y` and then `M-y` repeatedly, until the text that we want is yanked.

- Insert the contents of a file: Move the point to the desired place and type `C-x i` and the path of the file. Alternatively, give the command `M-x insert-file`.

- Insert the contents of a buffer: We can insert the contents of a whole buffer at a point by giving the command `M-x insert-buffer`.

- Replace text: We can replace text interactively with the command `M-x query-replace`, then type the string we want to replace, and then the replacement string. Then, we will be asked whether we want the change to be made and we can answer by typing `y` (yes),

[26] If it is a two button mouse, try clicking the left and right buttons simultaneously.

`n` (no), `q` (quit the replacements). A `,` (comma) makes only one replacement and quits (useful if we know that this is the last change that we want to make). If we are confident, we can change all string in a buffer, no questions asked, by giving the command `M-x replace-string`.

- Change case: We can change the case in the words of a region with the commands `M-x upcase-region`, `M-x capitalize-region` and `M-x downcase-region`. Try it.

We note that cutting and pasting can be made between different windows of the same or different buffers.

### 1.3.5 Windows

Sometimes it is very convenient to edit one or more different buffers in two or more windows. The term "windows" in Emacs refers to regions of the same Emacs desktop window. In fact, a desktop window running an Emacs session is referred to as a *frame* in the Emacs jargon. Emacs can split a frame in two or more windows, horizontally or/and vertically. Study figure 1.5 on page 69 for details. We can also open a new frame and edit several buffers simultaneously[27]. We can manipulate windows and frames as follows:

- Position the point at the center of the window and clear the screen from garbage: `C-l` (careful: `l` not `1`).
- Split a window in two, horizontally: `C-x 2`.
- Split a window in two, vertically: `C-x 3`.
- Delete all other windows (remain only with the current one): `C-x 1`.
- Delete the current windows (the others remain): `C-x 0`.
- Move the cursor to the other window: `Mouse-1` or `C-x o`.
- Change the size of window: Use `Drag-Mouse-1` on the line separating two windows (the mode line). Use `C-^`, `C-}` for making a change of the horizontal/vertical size of a window respectively.

[27]Be careful not to start a new Emacs session each time that all you need is a new frame. A new Emacs process takes time to start, binds computer resources and does not communicate with a different Emacs process.

- Create a new frame: `C-x 5 2`.
- Delete a frame: `C-x 5 0`.
- Move the cursor to a different frame: With `Mouse-1` or with `C-x 5 o`.

You can have many windows in a dumb terminal. This is a blessing when a dekstop environment is not available. Of course, in that case you cannot have many frames.

### 1.3.6 Files and Buffers

- Open a file: `C-x C-f` or `M-x find-file`.
- Save a buffer: `C-x C-s` or `M-x save buffer`. With `C-x C-c` or `M-x save-buffers-kill-emacs` we can also exit Emacs. From the menu: `File`→`Save`. From the toolbar: click on the save icon.
- Save buffer contents to a different file: `C-x C-w` or `M-x write-file`. From the menu: `File`→`Save As`. From the toolbar: click on the "save as" icon.
- Save all buffers: `C-x s` or `M-x save-some-buffers`.
- Connect a buffer to a different file: `M-x set-visited-filename`.
- Kill a buffer: `C-x k`.
- Change the buffer of the current window: `C-x b`. Also, use the menu `Buffers`, then choose the name of the buffer.
- Show the list of all buffers: `C-x C-b`. From the menu: `Buffers` → `List All Buffers`. By typing `Enter` next to the name of the buffer, we make it appear in the window. There are several buffer administration commands. Learn about them by typing `C-h m` when the cursor is in the Bufer List window.
- Recover data from an edited buffer: If Emacs crashed, do not despair. Start a new Emacs and type `M-x recover-file` and follow the instructions. The command `M-x recover-session` recovers all unsaved buffers.

- Backup files: When you save a buffer, the previous contents of the file become a backup file. This is a file whose path is the same as the original's file with a ~ appended in the end. For example a file `test.f90` will have as a backup the file `test.f90~`. Emacs has version control, and you can configure it to keep as many versions of your edits as you want.

- Directory browsing and directory administration commands: `C-x d` or `M-x dired`. You can act on the files of a directory (open, delete, rename, copy etc) by giving appropriate commands. When the cursor is in the `dired` window, type `C-h m` to read the relevant documentation.

### 1.3.7 Modes

Each buffer can be in different *modes*. Each mode may activate different commands or editing environment. For example each mode can color keywords relevant to the mode and/or bind keys to different commands. There exist *major modes*, and a buffer can be in only one of them. There are also *minor modes*, and a buffer can be in one or more of them. Emacs activates major and minor modes by default for each file. This usually depends on the filename but there are also other ways to control this. The user can change both major and minor modes at will, using appropriate commands.

Active modes are shown in a parenthesis on the mode line (see figures 1.3 and 1.5.

- `M-x f90-mode`: This mode is of special interest in this book since we will edit a lot of Fortran code. We need it activated in buffers that contain a Fortran program and its most useful characteristics are automatic code alignment by pressing the key `TAB`, the coloring of Fortran commands, variables and other structural constructs (subroutines, if statements, do loops, variable declarations, statement labels etc). Another interesting function is the one that comments out a whole region of code, as well as the inverse function.

- `M-x c-mode`: For files containing programs written in the C language. Related modes are the `c++-mode`, `java-mode`, `perl-mode`, `awk-mode`, `python-mode`, `makefile-mode`, `octave-mode`, `gnuplot-mode`, `mathematica-mode` and others.

- `latex-mode`: For files containing LaTeX text formatting commands.

- `text-mode`: For editing simple text files (`.txt`).
- `fundamental-mode`: The basic mode, when one that fits better doesn't exist...

Some interesting minor modes are:

- `M-x auto-fill-mode`: When a line becomes too long, it is wrapped automatically. A related command to do that for the whole region is `M-x fill-region`, and for a paragraph `M-x fill-paragraph`.
- `M-x overwite-mode`: Instead of inserting characters at the point, overwrite the existing ones. By giving the command several times, we toggle between activating and deactivating the mode.
- `M-x read-only` mode: When visiting a file with valuable data that we don't want to change by mistake, we can activate this mode so that changes will not be allowed by Emacs. When we open a file with the command `C-x C-r` or `M-x find-file-read-only` this mode is activated. We can toggle this mode on and off with the command `C-x C-q (M-x toggle-read-only)`. See the mode line of the buffer `jack.c` in figure 1.5 which contains a string %%. By clicking on the %% we can toggle the `read-only` mode on and off.
- `flyspell-mode`: Spell checking as we type.
- `font-lock-mode`: Colors the structural elements of the buffer which are defined by the major mode (e.g. the commands of a Fortran program).

In a desktop environment, we can choose modes from the menu of the mode line. By clicking with `Mouse-3` on the name of a mode we are offered options for (de)activating minor modes. With a `Mouse-1` we can (de)activate the `read-only` mode with a click on `:%%` or `:--` respectively. See figure 1.5.

### 1.3.8 Emacs Help

Emacs' documentation is impressive. For newbies, we recommend to follow the mini course offered by the Emacs tutorial. You can start the tutorial by typing `C-h t` or select `Help` → `Emacs Tutorial` from the menu. Enjoy... The Emacs man page (give the `man emacs` command in the command line) will give you a summary of the basic options when calling Emacs from the command line.

A quite detailed manual can be found in the Emacs `info` pages[28]. Using `info` needs some training, but using the Emacs interface is quite intuitive and similar to using a web browser. Type the command `C-h r` (or choose `Help`→`Emacs Tutorial` from the menu) and you will open the front page of the emacs manual in a new window. By using the keys `SPC` and `Backspace` we can read the documentation page by page. When you find a link (similar to web page hyperlinks), you can click on it in order to open to read the topic it refers to. Using the navigation icons on the toolbar, you can go to the previous or to the next pages, go up one level etc. There are commands that can be given by typing single characters. For example, type `d` in order to jump to the main `info` directory. There you can find all the available manuals in the `info` system installed on your computer. Type `g (emacs)` and go to the top page of the Emacs manual. Type `g (info)` and read the `info` manual.

Emacs is structured in an intuitive and user friendly way. You will learn a lot from the names of the commands: Almost all names of Emacs commands consist of whole words, separated by a hyphen "-", which almost form a full sentence. These make them quite long sometimes, but by using auto completion of their names this does not pose a grave problem.

- `auto completion`: The names of the commands are auto completed by typing a `TAB` one or more times. E.g., type `M-x` in order to go to the minibuffer. Type `capi[TAB]` and the command autocompletes to `capitalize-`. By typing `[TAB]` for a second time, a new window opens and offers the options for completing to two possible commands: `capitalize-region` and `capitalize-word`. Type an extra `r[TAB]` and the command auto completes to the only possible choice `capitalize-region`. You can see all the commands that start with an `s` by typing `M-x s[TAB][TAB]`. Sure, there are many... Click on the `*Completions*` buffer and browse the possibilities. A lot will become clear just by reading the names of the commands. By typing `M-x [TAB][TAB]`, all available commands will appear in your buffer!

- `keyboard shortcuts`: If you don't remember what happens when you type `C-s`, no problem: Type `C-h k` and then the ... forgotten key sequence `C-s`. Conversely, have you forgotten what is the keyboard

[28] If you prefer books in the form of `PDF` visit the page `www.gnu.org/software/emacs` and click on `Documentation`. You will find a 600 page book that has almost everything!

shortcut of the command `save-buffer`? Type `C-h w` and then the command.

- `functions`: Are you looking for a command, e.g. `save-something -I-forgot`? Type `C-h f` and then `save-[TAB]` in order to browse over different choices. Use `Mouse-2` in order to select the command you are interested in, or type and complete the rest of its name (you may use `[TAB]` again). Read about the function in the `*Help*` buffer that opens.
- `variables`: Do the same after typing `C-h v` in order to see a variable's value and documentation.
- command `apropos`: Have you forgotten the exact name of a command? No problem... Type `C-h a` and a keyword. All commands related to the keyword you typed will appear in a buffer. Use `C-h d` for even more information.
- `modes`: When in a buffer, type `C-h m` and read information about the active modes of the buffer.
- `info`: Type `C-h i`
- Have you forgotten everything mentioned above? Just type `C-h ?`

### 1.3.9 Emacs Customization

You can customize everything in Emacs. From key bindings to programming your own functions in the Elisp language. The most common way for a user to customize her Emacs sessions, is to put all her customization commands in the file `~/.emacs` in her home directory. Emacs reads and executes all these commands just before starting a session. Such a `.emacs` file is given below:

```
; Define F1 key to save the buffer
(global-set-key [f1]     'save-buffer)
; Define Control-c s to save the buffer
(global-set-key "\C-cs" 'save-some-buffers)
; Define Meta-s (Alt-s) to interactively search forward
(global-set-key "\M-s"  'isearch-forward)
; Define M-x is         to interactively search forward
(defalias       'is     'isearch-forward)
; Define M-x fm         to set fortran-mode for the buffer
(defun fm()    (interactive) (f90-mode))
```

```
; Define M-x sign        to sign my name
(defun sign()  (interactive) (insert "K. N. Anagnostopoulos"))
```

Everything after a ; is a comment. Functions/commands are enclosed in parentheses. The first three ones bind the keys `F1`, `C-c s` and `M-s` to the commands `save-buffer`, `save-some-buffers` and `isearch-forward` respectively. The next one defines an *alias* of a command. This means that, when we give the command `M-x is` in the minibuffer, then the command `isearch-forward` will be executed. The last two commands are the definitions of the functions `(fm)` and `(sign)`, which can be called interactively from the minibuffer.

For more complicated examples google "emacs .emacs file" and you will see other users' `.emacs` files. You may also customize Emacs from the menu commands `Options`→`Customize Emacs`. For learning the Elisp language, you can read the manual "Emacs Lisp Reference Manual" found at the address
`www.gnu.org/software/emacs/manual/elisp.html`

## 1.4 The Fortran Programming Language

In this section, we give a very basic introduction to the Fortran programming language. This is not a systematic exposition and you are expected to learn what is needed in this book *by example*. So, please, if you have not already done it, get in front of a computer and *do* what you read. You can find many good tutorials and books introducing Fortran in a more complete way in the bibliography.

### 1.4.1 The Foundation

The first program that one writes when learning a new programming language is the "Hello World!" program. This is the program that prints "Hello World!" on your screen:

```
program hello

!print a message to the world:
 print *, 'Hello World!'  !this is a comment

end program hello
```

Commands, or *statements*, in Fortran are strings of characters separated by blanks ("words") that we are allowed to write from the 1st to the 132nd column of a file. Each line starts a new command[29]. We can put more than one command on each line by separating them with a semicolon (`;`). Everything after an exclamation mark (`!`) is a comment. Proliferation of comments is necessary for documenting our code. Good documentation of our code is an integral part of programming. If the code is planned to be read by others, or by us at a later time, make sure to explain in detail what each line is supposed to do. You and your collaborators will save a lot of time in the process of debugging, improving and extending your code.

The main entry to the program is defined by the command `program name`, where `name` can be any string of alphanumeric characters and an underscore. When the program runs, it starts executing commands at this point. The end of the program, as well as of any other program unit (functions, subroutines, modules), is defined by the line `end program name`.

The first (and only) command given in the above program is the `print` command. It prints the string of characters "Hello World!" to the `stdout`. The "`*,`" is part of the syntax and it is not printed, of course. Fortran does not distinguish capital from small letters, so we could have written `PRINT`, `Print`, `prINt`, ... A string of characters in Fortran is enclosed in single or double quotes (`'Hello World!'` or `"Hello World!"` is equivalent).

In order to execute the commands in a program, it is necessary to *compile* it. This is a job done by a program called the *compiler* that translates the human language programming statements into binary commands that can be loaded to the computer memory for execution. There are many Fortran compilers available, and you should learn which compilers are available for use in your computing environment. Typical names for Fortran compilers are `gfortran, f90, ifort, g95, ....` You should find out which compiler is best suited for your program and spend time reading its documentation carefully. It is important to learn how to use a new compiler so that you can finely tune it to optimize the performance of *your* program.

We are going to use the open source and freely available compiler `gfortran`, which can be installed on most popular operating systems[30]. The compilation command is:

[29] It is possible to break long lines by putting a `&` at the end of each broken line and continue the same command in the next one. More on that later.

[30] `http://gcc.gnu.org/fortran/`

```
> gfortran hello.f90 -o hello
```

The switch `-o` defines the name of the executable file, which in our case is `hello`. If the compilation is successful, the program runs with the command:

```
> ./hello
 Hello world!
```

Now, we will try a simple calculation. Given the radius of a circle we will compute its length and area. The program can be found in the file `area_01.f90`:

```
program circle_area

 PI = 3.141593
 R  = 4.0
 print *,'Perimeter= ',2.0*PI*R
 print *,'Area=      ',PI*R**2

end program circle_area
```

The first two commands define the values of the *variables* `PI` and `R`. These variables are of type `REAL`, which are *floating point* numbers. Fortran has *implicit rules* that can be used to define the type of variables. By default, variables whose name starts with `i, j, k, l, m` and `n` are of `INTEGER` type. These are exact whole numbers. All other variables are of type `REAL`[31]. We can override these implicit rules by explicitly declaring the type of a variable or by changing the implicit rules with the use of the `implicit` statement. The following two commands have two effects: Computing the length $2\pi R$ and the area $\pi R^2$ of the circle and printing the results. The expressions `2.0*PI*R` and `PI*R**2` are evaluated before being printed by the `print` command. The multiplication and raising to a power operators are `*` and `**`, respectively. Note the explicit decimal points at the constants `2.0` and `4.0`. If we write `2` or `4` instead, then these are going to be constants of the `INTEGER` type and by using them

[31]Don't confuse `REAL` variables with the real numbers. `REAL` variables take values that are finite approximations to real numbers and take values that are a subset of the rational numbers. This approximation becomes better with increasing the amount of memory allocated to `REAL`s. In most computing environments, `REAL`s are allocated 4 or 8 bytes of memory, in which case they approximate real numbers with, more or less, 7 or 17 significant digits, respectively.

the wrong way we may obtain surprising results[32]. We compile and run the program with the commands:

```
> gfortran area_01.f90 -o area
> ./area
 Perimeter=     25.13274
 Area=          50.26548
```

We will now try a process that repeats itself for many times. We will calculate the length and area of 10 circles of different radii $R_i = 1.28 + i$, $i = 1, 2, \ldots, 10$. We will store the values of the radii in an *array* `R(10)` of the `REAL` type. The code can be found in the file `area_02.f90`:

```
program circle_area

 dimension R(10)

 PI = 3.141593
 R(1) = 2.28
 do i=2,10
  R(i) = R(i-1) + 1.0
 enddo

 do i = 1,10
  perimeter = 2*PI*R(i)
  area      = PI*R(i)**2
  print *,i,') R= ',R(i),' area= ',area,&
          ' perimeter= ',perimeter
 enddo

end program circle_area
```

The command `dimension R(10)` defines an array of length 10. This way, the elements of the array are referred by an index that takes value from 1 to 10. For example `R(4)` is the fourth element of the array.

Between the lines

```
 do  i = 2, 10
  ...
 enddo
```

[32]Try adding the command `print *,2/4, 2.0/4.0` and check the results.

we can write commands that are repeatedly executed while the INTEGER variable i takes values from 2 to 10 with increasing step[33] equal to 1. The command:

```
R(i) = R(i-1) + 1.0
```

defines the i-th radius to have a value which is larger by the (i-1)-th by 1. For the loop to work correctly, we must define the initial value of R(1), otherwise the final result is undefined[34]. The second loop uses the defined R-values in order to do the computation and printing of the results.

Now, we will write an interactive version of the program. Instead of *hard coding* the values of the radii, we will interact with the user asking her to give her own values. The program will read the 10 values of the radii from the standard input (stdin). The program can be found in the file area_03.f90:

```
program circle_area

 implicit none

 integer,parameter     :: N=10
 real   ,parameter     :: PI=3.141593
 real   ,dimension(N)  :: R
 real                  :: area,perimeter
 integer               :: i

 do i=1,N
  print*,'Enter radius of circle: '
  read *, R(i)
  print*,'i= ',i,' R(i)= ',R(i)
 enddo

 open(UNIT=13,FILE='AREA.DAT')

 do i = 1,N
  perimeter = 2*PI*R(i)
  area      = PI*R(i)**2
  write(13,*)i,') R= ',R(i),' area= ',area,&
       ' perimeter= ',perimeter
 enddo
```

[33]The step can change by adding one more entry to the do line: do i=0,12,4 runs the loop for i=0,4,8,12, whereas do i=10,6,-2 for i=10,8,6.

[34]That means that different compilers and/or runs can give different results.

```
 close(13)

end program circle_area
```

The first statement in the above program is `implicit none`! This statement deactivates the implicit rules of Fortran, and the programmer is *obliged* to declare all variables in a program unit. It is highly recommendable that you *always* use this option... You might spend a little more time typing the declarations, but this effort cannot be compared to the pain looking for bugs due to typos in the names of variables[35]! We will follow this practice throughout the book.

The declarations of the variables follow this statement. The variables `N` and `i` are declared to be of the `INTEGER` type, whereas the variables `PI, area, perimeter` and `R(N)` are declared to be of the `REAL` type. The variables `PI` and `N` are specified to be parameters. Parameters are given specific values which cannot be changed during the execution of the program.

The array elements `R(i)` are read using the command `read`:

```
  read *, R(i)
```

The command `read` reads from the `stdin`. The user types the values at the terminal and then presses `[Enter]`. We can read more than one variables with one `read` command.

In order to print data to a file, we have to connect it to a *unit*. Each unit is represented by any number between 0 and 99. Some numbers are reserved for special units[36]. The connection of a unit to a file is done with the `open` command. When this is done, we can write to the file with the command[37] `write(n,*)`, where `n` is the unit number. When we are done writing to a file we should use the command `close(n)`. Then the unit number is available to be used for a different file. The flow of commands is like

```
open(UNIT=13,FILE='AREA.DAT')
...
write(13,*) ....
...
```

[35]Can you see the difference between the names `pl1` and `p11`?

[36]E.g., 5 is the `stdin`, 6 is the `stdout` and 0 is the `stderr`.

[37]Try to see what happens when you write to a unit what has not been connected to a file via an `open` command!

```
close(13)
```

The name of the file is determined by the option `FILE='AREA.DAT'` of the `open` statement. Uppercase or lowercase characters in the filename make a difference. The option `FILE='path'` can use any valid path in the filesystem, provided that we have the necessary permissions.

The line

```
write(13,*)i,') R= ',R(i),' area= ',area,&
     ' perimeter= ',perimeter
```

shows us how to continue a line containing a long statement to the next one. We place a `&` at the end of the line and then continue writing the statement to the next. This can happen up to 39 times.

The next step will be to learn how to define and use functions and subroutines. The program below shows how to define a `subroutine` `area_of_circle`, which computes the length and area of a circle of given radius. The following program can be found in the file area_04.f90:

```
program circle_area

 implicit none

 integer,parameter    :: N=10
 real   ,parameter    :: P=3.141593
 real   ,dimension(N):: R
 real                 :: area,perimeter
 integer              :: i

 do i=1,N
  print*,'Enter radius of circle: '
  read *, R(i)
  print*,'i= ',i,' R(i)= ',R(i)
 enddo

 open(UNIT=13,FILE='AREA.DAT')

 do i = 1,N
  call area_of_circle(R(i),perimeter,area)
  write(13,*)i,') R= ',R(i),' area= ',area,&
         ' perimeter= ',perimeter
 enddo

 close(13)
```

```
end program circle_area

subroutine area_of_circle(R,L,A)
 implicit none
 real            :: R,L,A
 real,parameter :: PI = 3.141593 , PI2 = 2.0*PI

 L= PI2*R
 A= PI*R*R

 return

end subroutine area_of_circle
```

The calculation of the length and the area of the circle is performed by the `call` to the subroutine:

```
  call area_of_circle(R(i),perimeter,area)
```

The command `call` calls a subroutine and transfers the control of the program within the subroutine. The above subroutine has the *arguments* `(R(i),perimeter,area)`. The argument `R(i)` is an input variable. It provides the necessary data to the subroutine in order to perform its computation. The arguments `perimeter` and `area` are intended for output. Upon return of the subroutine to the main program, they store the result of the computation. The user of a subroutine must learn how to use its arguments in order to be able to call it in her program. These must be documented carefully by the programmer of the subroutine.

The actual program executed by the subroutine is between the lines:

```
subroutine area_of_circle(R,L,A)
 ...
end subroutine area_of_circle
```

The arguments `(R,L,A)` must be declared in the subroutine and need not have the same names as the ones that we use when we call it. A change of their values within the subroutine will change the values of the corresponding variables in the calling program[38]. Therefore, the statements `L=PI2*R` and `A=PI*R*R` change the values of the variables `perimeter` and `area` to the desired values. The command `return` returns the control to the calling program. The parameters `PI` and `PI2` are "private" to the

[38]We say that variables in Fortran are passed to subroutines *by reference* and not *by value* as in C.

subroutine. Their names and values are invisible outside the subroutine. Similarly, the variables `i`, `N`, ..., defined in the main program, are invisible within the subroutine.

We summarize all of the above in a program `trionymo.f90`, which computes the roots of a second degree polynomial:

```
! ==============================================================
! Program to compute roots of a 2nd order polynomial
! Tasks: Input from user,logical statements,
!        use of functions,stop
!        Accuracy in floating point arithmetic
!        e.g. IF(x.eq.0.0)
!
! Tests: a,b,c= 1  2  3 D=   -8
!        a,b,c= 1 -8 16 D=    0  x1=   4
!        a,b,c= 1 -1 -2 D=    9. x1=   2. x2=  -1.
!        a,b,c= 2.3 -2.99 -16.422 x1=   3.4 x2=  -2.1
! But:   6.8(x-4.3)**2 = 6.8 x**2 -58.48*x+125.732
!        a,b,c= 6.8 -58.48 125.73199
!        D= 0.000204147349  x1=   4.30105066 x2=   4.29894924
!        a,b,c= 6.8 -58.48 125.732, D=   -0.000210891725 < 0!!
! ==============================================================
program trionymo
 implicit none
 real :: a,b,c,D
 real :: x1,x2
 real :: Discriminant

 print*,'Enter a,b,c:'
 read *,a,b,c

! Test if we have a well defined polynomial of 2nd degree:
 if( a .eq. 0.0) stop 'trionymo: a=0'

! Compute the discriminant (= diakrinousa)
 D = Discriminant(a,b,c)
 print *, 'Discriminant: D=  ',D


! Compute the roots in each case: D>0, D=0, D<0 (no roots)
 if(D .gt. 0.0 )then
  call roots(a,b,c,x1,x2)
  print *,'Roots:        x1= ',x1,' x2= ',x2
 else if (D .eq. 0.0) then
  call roots(a,b,c,x1,x2)
  print *,'Double Root:  x1= ',x1
 else
```

```
  print *,'No real roots'
 endif

end program trionymo
! ==============================================================
! This is the function that computes the discriminant
! A function returns a value. This value is assigned with the
! statement:
! Discriminant = <value>
! i.e. we simply assign anywhere in the program a variable with
! the name  of the function.
! ==============================================================
real function Discriminant(a,b,c)
 implicit none
 real :: a,b,c

 Discriminant = b**2 - 4.0 * a * c

end function Discriminant
! ==============================================================
! The subroutine that computes the roots.
! ==============================================================
subroutine roots(a,b,c,x1,x2)
 implicit none
 real :: a,b,c
 real :: x1,x2
 real :: D, Discriminant

 if(a .eq. 0.0) stop 'roots: a=0'

 D = Discriminant(a,b,c)
 if(D.ge.0.0)then
  D = sqrt(D)
 else
  print *,'roots: Sorry, cannot compute roots, D<0=',D
  stop
 endif

 x1 = (-b + D)/(2.0*a)
 x2 = (-b - D)/(2.0*a)

end subroutine roots
```

The program reads the coefficients of the polynomial $ax^2 + bx + c$. After a check whether $a \neq 0$, it computes the discriminant $D = b^2 - 4ac$ by calling the `function Discriminant(a,b,c)`. The only difference between a function and a subroutine is that the first one returns a value of a given type. We don't need to use the command `call` in order to run

the commands of a function, this is done by computing its value in an expression. The type of the value returned must be declared both in the program that uses the function (`real :: Discriminant`) and at the entry point of its program unit (`real function Discriminant(a,b,c)`). The value returned to the calling program is the one assigned to the variable that has the same name as the function:

```
real function Discriminant(a,b,c)
 ...
 Discriminant = b**2 - 4.0 * a * c
 ...
end function Discriminant
```

Notice the use of the comparison operators `.gt.` (strictly greater than) and `.eq.` (equal to)[39]:

```
 if(D .gt. 0.0 )then
  ...
 else if (D .eq. 0.0) then
  ...
 else
  ...
 endif
```

### 1.4.2 Details

You may skip this paragraph during a first reading of the book. It is intended mainly to be a reference when reading the later chapters.

There are more types of variables built in Fortran. In the program listed below, we show how to use `CHARACTER` variables, floating point numbers of double precision `REAL(8)` and complex numbers of single and double precision, `COMPLEX` and `COMPLEX(8)` respectively:

```
program f90_vars
 implicit none

 character(100) :: string

 real(4)    ::   x !single precision, same as real :: x
 real(8)    ::  x8 !equivalent to: double precision x8
```

[39]Other operators are `.lt.`, `.ge.` `.le.` (strictly less, greater or equal, less or equal), `.ne.` (not equal) and `.or.`, `.and.`, `.not.` (logical or, and and negation).

```
!real(16)   :: x16 !may not be supported by all compilers
!Complex Numbers:
 complex(4) :: z   !single precision, same as complex :: z
 complex(8) :: z8  !double precision

!A string array:
 string = 'Hello World!' !string smaller size, leaves blanks
                                          !TRIM: trim blanks
 print *,'A string ::',      string, '::',TRIM(string),'::'
 print *,'join them::',      string   //       string ,'::'
 print *,'join them::', TRIM(string)  //  TRIM(string),'::'
!Reals with increasing accuracy: Determine PI=3.14159...
 x   = 4.0  *atan(1.0  )
!Use D for double    precision exponent
 x8  = 4.0D0*atan(1.0D0)
!Use Q for quadriple precision exponent
!x16 = 4.0Q0*atan(1.0Q0)
 print *,'x4= ',x,' x8= ',x8 !,' x16= ',x16
 print *,'x4: ',range(x ),precision(x ),EPSILON(x ),&
          TINY(x ),HUGE(x )
 print *,'x8: ',range(x8),precision(x8),EPSILON(x8),&
          TINY(x8),HUGE(x8)

!Complex numbers: single precision
 z = (2.0,1.0)*cexp((3.0,-1.0))
 print *,'z= ',z,' Re(z)= ',REAL(z),' Im(z)= ',IMAG(z),&
      ' |z|= ',ABS(z),' z*= ',CONJG(z)

!Complex numbers: double precision
 z8 = (2.0D0,1.0D0)*cdexp((3.0D0,-1.0D0))
 print *,'z= ',z8,' Re(z)= ',DBLE(z8),' Im(z)= ',DIMAG(z8),&
      ' |z|= ',CDABS(z8),' z*= ',DCONJG(z8)
 print *,'z4: ',range(z ),precision(z )
 print *,'z8: ',range(z8),precision(z8)

end program f90_vars
```

Some interesting points of the program above are:

- The number `K` in the declaration `REAL(K):: x` refers to the number of bytes allocated to the variable `x`. For `K=4` we have single precision (same as `REAL`), for `K=8` double precision and for `K=16` quadruple precision. The latter is not always available. In the declarations `COMPLEX(K)`, `K` refers to the number of bytes allocated to the real and imaginary parts of the complex number.

- We always use the exponent notation `D` in double precision constants, even if the exponent 0. Otherwise the constants are of single

precision and we loose the desired accuracy.

- When we want to state the precision of the return value of an intrinsic function explicitly, we usually add a `d` at the beginning of its name (e.g. `exp`→`dexp`, `ABS`→`DABS`. When we want to use the complex version of a function, we usually add a `c` at the beginning of its name (e.g. `exp`→`cexp`, `ABS`→`CABS`). Modify the program in order to achieve higher accuracy in the calculation of $\pi$ and $z = (2+i)e^{3-i}$, by using double precision variables.

- The maximum number of characters in the `CHARACTER` variable `string` is 100, and this is declared by the statement `CHARACTER(100)`.

- When we print a `CHARACTER` variable, *all* its characters are printed, including trailing blanks. This is very annoying and we can use the function `TRIM` in order to remove them.

- The operator `//` joins two `CHARACTER` variables or constants. Notice the effect of the function `TRIM` in the above program.

Another important point to discuss is how to be able to access the same variables from different program units. So far, we simply mentioned that variables have a *scope* within each function and subroutine. If we wish to have access to the same location of memory[40] from different program units, then we use the `COMMON` statement which defines a *common block*. See the following example:

```
! ----------------------------------------
program f90_common
 implicit none
 real :: k1=1.0,k2=1.0,k3=1.0
 common /CONSTANTS/k1,k2

 print *,'main: k1= ',k1,' k2= ',k2,' k3= ',k3
 call s1 !prints k1 and k2 but not k3
 call s2 !changes the value of k2 but not k3
 print *,'main: k1= ',k1,' k2= ',k2,' k3= ',k3

end program f90_common
! ----------------------------------------
subroutine s1()
```

[40]Common blocks are supposed to be obsolescent in Fortran and programmers are encouraged to avoid them and use modules instead. Due to their simplicity and popularity we will show their usage and also use them in this book.

```
 implicit none
 real k1,k2,k3
 common /CONSTANTS/k1,k2

 print *,'s1: k1= ',k1,' k2= ',k2,' k3= ',k3
end subroutine s1
! ----------------------------------------
subroutine s2()
 implicit none
 real k1,k2,k3
 common /CONSTANTS/k1,k2

 k2 = 2.0
 k3 = 2.0
end subroutine s2
```

The common block has the name `CONSTANTS` and we can refer to it from any program unit. Each program unit that uses this common block must use the same declaration, although the names of variables are allowed to be different. The common block `CONSTANTS` points to the same location in the computer memory, where we expect to find the values of two real variables. These variables (`k1` and `k2`) are used and have their values changed in the subroutines `s1` and `s2`. The variable `k3`, is a different variable in each program unit. The program prints

```
 main: k1=    1.000000   k2=    1.000000   k3=    1.000000
 s1:   k1=    1.000000   k2=    1.000000   k3=  -2.8117745E-05
 main: k1=    1.000000   k2=    2.000000   k3=    1.000000
```

One of the weaknesses of Fortran is that it does not have a convenient control for Input/Output (I/O). For complicated I/O and text manipulation we will use other programs that can do a better job, like `awk`, `perl`, shell scripting, or programs written in `C/C++`. It is important to know some details about I/O commands in Fortran, mainly the specifications that control the accuracy of printed floating point numbers. So far, I/O commands, like `print`, `write`, `read`, used a `*` in order to control the printing of numbers. But we can replace the `*` with explicit format directives as follows:

```
program f90_format1
 implicit none
 integer              :: i
 real                 :: x
 real , dimension(10) :: a
```

```
 real(8)              :: x8

 i  = 123456
 x  = 2.0  *atan2(1.0,0.0)
 print '(A5,I6,F12.7)','x,i= ',i,x
 x8 = 2.0D0*atan2(1.0D0,0.0D0)
 write(6,'(F18.16,E24.17,G24.17,G24.17)') x8,x8,&
      1.0D15*x8,1.0D18*x8
 write(6,'(3F20.16)') x8,x8/2.0,cos(x8)
 write(6,'(200F12.6)')(a(i), i=1,10)
end program f90_format1
```

Note the parentheses within the single quotes: `(A5,I6,F12.7)` is a format directive for the print statement. The `A` is for printing a `CHARACTER`, the `I` for printing an `INTEGER` and the `F` for printing a floating point number. The numbers after the letter declare the number of spaces used for printing each one. Beware! If the printing space is not enough, Fortran will not print and you will find a series of `*` in place of the value of your result! Bummer... In order to estimate the number of spaces needed for a floating point number, you have to include the space taken by the decimal point, the sign, the exponent character, the sign of the exponent and the digits needed for the exponent. Plus a space to separate the numbers in between... So, be generous and give plenty of printing space. In the example shown above, `A5` denotes a character of 5 spaces, `I6` and integer of 6 spaces and `F12` a floating point number of 12 spaces. The decimal point in `F12.7` means that we want a floating point with the accuracy of 7 significant digits.

The format directive `(F18.16,E24.17,G24.17,G24.17)` shows how to print double precision variables. These provide an accuracy of 16-17 significant digits and there is no need for keeping more digits. The command `E` prints a number in scientific form with an exponent. The command `G` prints the exponent when it is needed. The numbers before the letters denote multiplicity. Therefore `3F20.16` instructs the printing of 3 floating point numbers by reserving 20 spaces and using 16 significant digits for each one of them.

The command `write(6,'(200F12.6)')(a(i), i=1,10)` shows how to print a large array using an implicit loop. We used many more spaces than actually needed (`200F12.16`) which is OK. If the array gets larger by increasing the range of `i`, then we will have enough room for printing in the same line. The program prints (we have folded the long line in order to make it visible):

```
x,i= 123456    3.1415927
3.1415926535897931 0.31415926535897931E+01   3141592653589793.0
                                          0.31415926535897933E+19
  3.1415926535897931   1.5707963267948966  -1.0000000000000000
    0.000000     0.000000     0.000000  ....
```

We can organize the format commands by using the `FORMAT` statement. Then, we use *labeled statements* in order to refer to them. Labels are numbers put in the beginning of a line which should be unique to a program unit and are within the range 1-99999. We can transfer the control of the program to such a line with a `goto` command or by using the label in the I/O statements `print`, `write` and `read` as in the example shown below:

```
program f90_format2
 implicit none
 integer i
 real    x, a(10)
 real*8  x8

 i  = 123456
 x  = 2.0  *atan2(1.0,0.0)
 print 100,'x,i= ',i,x
 x8 = 2.0D0*atan2(1.0D0,0.0D0)
 write(6,123) x8,x8,&
      1.0D15*x8,1.0D18*x8
 write(6,4444) x8,x8/2.0,cos(x8)
 write(6,9999)(a(i), i=1,10)
100  FORMAT(A5,I6,F12.7)
123  FORMAT(F18.16,E24.17,G24.17,G24.17)
4444 FORMAT(3F20.16)
9999 FORMAT(200F12.6)
end program f90_format2
```

The reader should also study the Fortran intrinsic functions shown in table 1.2, page 71.

### 1.4.3 Arrays

You may skip this section during the first reading of this book. It will be useful to come back here later.

Arrays are related data of the same type which can be accessed by using one or more indices. For example, after a declaration `real, dimension(10) :: A`, the expressions

```
A(1), A(2), ... , A(10)
```

refer to its 10 real values. The indices can be integer expressions, for example

```
A(i), B(2*i+3), C(INT(x+y(j)))
```

where in the last case we used the integer value of the intrinsic function `INT(x)`, which returns the integer part of `x`. Note that, arrays and functions enclose indices and arguments between parentheses `(...)` which are of the same style, and the compiler must look at their declarations in order to tell the difference. Examples of array declarations are

```
real, dimension(10) :: a,b
real, dimension(20) :: c,d
```

which declare the arrays `a, b, c, d`, which are of the real kind, with elements `a(1) ... a(10)`, `b(1) ... b(10)`, `c(1) ... c(20)` and `d(1) ... d(20)`. An equivalent declaration is

```
real :: a(10), b(10), c(20), d(20)
```

or

```
integer, parameter      :: n1 = 10, n2 = 20
real,    dimension(n1)  :: a, c(n2)
real                    :: b(n1), d(n2)
```

In the last form, we show how to use constant parameters for declaring the size of arrays. For the declarations shown above, the lower bound of all arrays is 1 and the upper bound for `a` and `b` is 10 and for `c` and `d` is 20. The upper and lower bound of arrays can be explicitly determined. The declarations

```
integer, parameter          :: n1 = 10, n2 = 20
real,    dimension(0:n1)    :: a
real,    dimension(-n1:n2)  :: c
```

define the real array `a` with 11 values `a(0) ... a(10)` and the array `c` with 31 values `c(-10) c(-9) ... c(-1) c(0) c(1) ... c(20)`.

The arrays shown above have *dimension* 1 and they are like vectors. We can declare arrays of more than one dimension. This means that we

need more than one indices in order to determine an array element[41]. Therefore, the declaration

```
integer, dimension(2,2)      :: a
```

defines an integer array with values `a(1,1)`, `a(1,2)`, `a(2,1)` and `a(2,2)`. The following declarations define two three dimensional real arrays `a` and `b`:

```
integer, parameter :: n1 = 10, n2 = 20, n3 = 2*n1+n2
real, dimension(n1,n2,n3)           :: a
real, dimension(-n1:n1,0:n2,13:n3) :: b
```

Some important definitions used in the bibliography are:

- **array**: Variables of the same type to which we refer with one or more indices. Variables with only one value are called **scalar**.
- An array's **dimension** has an **upper bound** and a **lower bound** which define the allowed range of index values. If the lower bound is omitted in a declaration, then it takes the value 1.
- The **rank** of an array is the number of its dimensions, i.e. the number of indices needed to determine its values.
- The **extent** of a dimension it the number of its elements. It is equal to (upper bound)-(lower bound)+1.
- The **size** of an array is the total number of its elements. For a one dimensional array, its size is equal to its extent, whereas for a multi dimensional one, it is equal to the product of the extents of all of its dimensions.
- The **shape** of an array is its rank and extents of all its dimensions.

The values of arrays can be set the same way as scalars:

```
integer :: i
real    :: a(4), b(2,2)

b(1,1) = 2.0 ; b(1,2) = 4.0
b(2,1) = 3.4 ; b(2,2) = 7.8
```

[41]Fortran allows up to seven indices in an array.

```
do i=1,4
 a(i) = 1.0
enddo
```

Alternatively we can use the name of the array as one object:

```
a = (/ 1.0, 2.0, 3.0, 4.0 /)
b = 0.0
```

The first line defines the values of an array by using an *array constructor*. The second line defines *all elements* of the array `b` to be equal to 0. This is an example of a very convenient feature of the Fortran language. If all the arrays in an expression are *conformable*, then we can use the intrinsic Fortran operations to act on whole arrays. Two arrays are conformable if they have the same shape or if one of them is a scalar. Therefore the program

```
integer :: i,j
real    :: x,y,a(10),b(10),c(4,4),d(4,4)

do i=1,10
 a(i) = b(i)
enddo

do j=1,4
 do i=1,4
  c(i,j) = x*d(i,j)+y
 enddo
enddo
```

is equivalent to

```
integer :: i,j
real    :: x,y,a(10),b(10),c(4,4),d(4,4)

a = b
c = x*d+y
```

Many Fortran intrinsic functions are *elemental*. This means that their arguments can be arrays, in which case the function acts on each array element separately. For example, the commands

```
integer :: i,j
real    :: x,y,a(10),b(10),c(4,4),d(4,4)
```

```
c = sin(d) + x*exp(-2.0*d)
call random_number(a)
```

set `c(i,j) = sin(d(i,j))+x*exp(-2.0*d(i,j))` for all `i` and `j`, and the elements of `a(i)` equal to a random number uniformly distributed in the interval $[0, 1)$. We should stress that in order for two arrays to be conformable, it is not necessary that they have the same lower and upper bounds. For example, the command `b=c*d` in the following program has the same effect as the `do loop`:

```
integer :: i
real    :: b(0:19), c(10:29), d(-9:10)

b = c*d

do i=1,20
 b(i-1) = c(i+9) * d(i-10)
enddo
```

In the following, we mention some useful functions that act on arrays. Assume that

```
real    :: a(-10:10), b(-10:10), c(10,10), d(10,10), e(10,10)
```

then

- `LBOUND(a)` and `UBOUND(a)` return the lower bound and the upper bound of the array `a`. In the above example `LBOUND(a)` = -10 and `UBOUND(a)` = 10.
- `c = TRANSPOSE(d)` sets `c(i,j)=d(j,i)`.
- `e = MATMUL(c,d)` sets the array `e` equal to the *matrix product* `c, d`. I.e. `e(i,j)=`$\sum_{k=1}^{10}$`c(i,k)*d(k,j)`. Be careful, the command `e=c*d` sets `e(i,j)=c(i,j)*d(i,j)`.
- `SUM(a)` computes the sum of all the elements of `a`. I.e. `SUM(a)` = $\sum_{i=-10}^{10}$`a(i)`
- `PRODUCT(a)` computes the product of all the elements of `a`. I.e. `PRODUCT(a)` = $\prod_{i=-10}^{10}$`a(i)`
- `DOT_PRODUCT(a,b)` computes the *inner product* of `a, b`. I.e. `DOT_PRODUCT(a,b)` = $\sum_{i=-10}^{10}$`a(i)*b(i)`

- `MAXVAL(a)` and `MINVAL(a)` return the maximum and minimum values in the array `a` respectively.

You can find more functions and documentation in the bibliography [9, 10]. In the following, we provide some information related to the Input/Output (I/O) of arrays. Input ("reading") and output ("writing") of array values can be done by reading and writing their elements in any order we want. In the example below, we read the array `a` and write the array `b` in two different ways:

```
integer :: i,j
real    :: a(4), b(2,2)

do i=1,4
 read *,a(i)
enddo
read *, (a(i), i=1,4)

do j=1,2
 do i=1,2
  print *,b(i,j)
 enddo
enddo
print *,( (b(i,j), i=1,2), j=1,2)
```

Inside the do loops, input and output is done one element per line from/to standard input/output. The commands `(a(i), i=1,4)` and `( (b(i,j) i=1,2), j=1,2)` are *implied do loops* and read/write from/to the same line. During input, if the number of values for `a` are exhausted, then the program tries to read values from the following line(s). Similarly, if the output of `b` exhausts the maximum number of characters per line, then the output continues in the next line[42]. Try it...

We can also preform I/O of arrays without explicit reference to their elements. In this case, the arrays are read/written in a specified order. For example, the program

```
real  :: a(4), b(2,2)

read  *, a
read  *, b
```

[42] If we want to force a long output to be written in one line, then we must replace the * by an explicit format directive, e.g. `print '(100I6)',( (c(i,j), i=1,10), j=1,10)`

```
print *, a,b
```

reads the values `a(1) a(2) a(3) a(4)` from the `stdin`. Then, it continues reading `b(1,1), b(2,1), b(1,2), b(2,2)` from the next line (record). Notice that the array `b` is read in a *column major* way. Printing `a` and `b`, will print `a(1) a(2) a(3) a(4)` and `b(1,1), b(2,1), b(1,2), b(2,2)` in two different records (also in column major mode).

Finally, we summarize some of the Fortran capabilities in array manipulation. More details can be found in the bibliography. Read the comments in the program for a partial explanation of each command:

```
program arrays
 implicit none
 integer :: i,j,n,m
 real    :: a(3), b(3,3), c(3,3)=-99.0, d(3,3)=-99.0, s
 integer :: semester(1000),grade(1000)
 logical :: pass(1000)
 !construct the matrix: use the RESHAPE function
 !|1.1 -1.2 -1.3|
 !|2.1  2.2 -2.3|
 !|3.1  3.2  3.3|
 b = RESHAPE((/  1.1,  2.1,  3.1, & !(notice rows<->columns)
                -1.2,  2.2,  3.2, &
                -1.3, -2.3,  3.3  /),(/3,3/))
 !same matrix, now exchange rows and columns: ORDER=(/2,1/)
 b = RESHAPE((/  1.2, -1.2, -1.3, &
                 2.1,  2.2, -2.3, &
                 3.1,  3.2,  3.3  /),(/3,3/),ORDER=(/2,1/))
 a = b(:,2) !a assigned the second column of b: a(i)=b(i,2)
 a = b(1,:) !a assigned the first  row    of b: a(i)=b(1,i)
 a = 2.0*b(:,3)+sin(b(2,:))!a(i)= 2*b(i,3)+sin(b(2,i))
 a = 1.0+2.0*exp(-a)+b(:,3)!a(i)= 1+2*exp(-a(i))+b(i,3)
 s = SUM(b)               !returns sum of all      elements of b
 s = SUM(b,MASK=(b.gt.0))!returns sum of positive elements of b
 a = SUM(b,DIM=1)     !each a(i) is the sum of the columns of b
 a = SUM(b,DIM=2)     !each a(i) is the sum of the rows    of b
 !repeat all the above using PRODUCT!
 !all instructions may be executed in parallel at any order!
 FORALL(i=1:3) c(i,i) = a(i) !set the diagonal of c
 !compute upper bounds of indices in b:
 n=UBOUND(b,DIM=1);m=UBOUND(b,DIM=2)
 !log needs positive argument, add a restriction ("mask")
 FORALL(i=1:n,j=1:m, b(i,j).gt.0.0 ) c(i,j) = log(b(i,j))
 !upper triangular part of matrix:
 !careful, j=i+1:m NOT permitted
 FORALL(i=1:n,j=1:m, i     .lt.  j ) c(i,j) = b(i,j)
 !each statement executed BEFORE the next one!
```

```
 FORALL(i=2:n-1,j=2:n-1)
  !all right hand side evaluated BEFORE the assignment
  !i.e., the OLD values of b averaged and then assigned to b
  b(i,j)=(b(i+1,j)+b(i-1,j)+b(i,j+1)+b(i,j-1))/4.0
  c(i,j)=1.0/b(i+1,j+1) !the NEW values of b are assigned
 END FORALL
 ! assignment but only for elements  b(i,j) which are not  0
 WHERE (b      .ne. 0.0)              c       = 1.0/b
 !MATMUL(b,c) is evaluated, then d is assigned the result only
 !at positions where b>0.
 WHERE (b      .gt. 0.0)              d       = MATMUL(b,c)
 WHERE (grade .ge. 5  )
  semester = semester + 1 !student's semester increases by 1
  pass     = .true.
 ELSEWHERE
  pass     = .false.
 END WHERE
end program arrays
```

The code shown above can be found in the file `f90_arrays.f90` of the accompanying software.

## 1.5 Gnuplot

Plotting data is an indispensable tool for their qualitative, but also quantitative, analysis. Gnuplot is a high quality, open source, plotting program that can be used for generating publication quality plots, as well as for heavy duty analysis of a large amount of scientific data. Its great advantage is the possibility to use it from the command line, as well as from shell scripts and other programs. Gnuplot is programmable and it is possible to call external programs in order manipulate data and create complicated plots. There are many mathematical functions built in gnuplot and a `fit` command for non linear fitting of data. There exist interactive *terminals* where the user can transform a plot by using the mouse and keyboard commands.

This section is brief and only the features, necessary for the following chapters, are discussed. For more information visit the official page of gnuplot `http://gnuplot.info`. Try the rich demo gallery at `http://gnuplot.info/screenshots/`, where you can find the type of graph that you want to create and obtain an easy to use recipe for it. The book [12] is an excellent place to look for many of gnuplot's secrets[43].

[43] A the time of the writing of this book, there was a very nice site `www.gnuplotting.org` which shows how to create many beautiful and complicated

You can start a gnuplot session with the `gnuplot` command:

```
> gnuplot

  G N U P L O T
  Version X.XX
  ....
  The gnuplot FAQ is available from www.gnuplot.info/faq/
  ....
Terminal type set to 'wxt'
gnuplot>
```

There is a welcome message and then a prompt `gnuplot>` is issued waiting for your command. Type a command an press `[Enter]`. Type `quit` in order to quit the program. In the following, when we show a prompt `gnuplot>`, it is assumed that the command after the prompt is executed from within gnuplot.

Plotting a function is extremely easy. Use the command `plot` and `x` as the independent variable of the function[44]. The command

```
gnuplot> plot x
```

plots the function $y = f(x) = x$ which is a horizontal line. In order to plot many functions simultaneously, you can write all of them in one line:

```
gnuplot> plot [-5:5][-2:4] x, x**2, sin(x),besj0(x)
```

The above command plots the functions $x$, $x^2$, $\sin x$ and $J_0(x)$. Within the square brackets `[:]`, we set the limits of the $x$ and $y$ axes, respectively. The bracket `[-5:5]` sets $-5 \leq x \leq 5$ and the bracket `[-2:4]` sets $-2 \leq y \leq 4$. You may leave the job of setting such limits to gnuplot, by omitting some, or all of them, from the respective positions in the brackets. For example, typing `[1:][:5]` changes the lower and upper limits of $x$ and $y$ and leaves the upper and lower limits unchanged[45].

---

plots.

[44]You can change the symbol of the independent variable. For example, the command `set dummy t` sets the independent variable to be `t`.

[45]By default, the $x$ and $y$ ranges are determined automatically. In order to *force* them to be automatic, you can insert a `*` in the brackets at the corresponding position(s). For example `plot [1:*][*:5]` sets the upper and lower limits of $x$ and $y$ to be determined automatically.

In order to plot data points $(x_i, y_i)$, we can read their values from files. Assume that a file `data` has the following numbers recorded in it:

```
# x   y1    y2
0.5  1.0  0.779
1.0  2.0  0.607
1.5  3.0  0.472
2.0  4.0  0.368
2.5  5.0  0.287
3.0  6.0  0.223
```

The first line is taken by gnuplot as a *comment line*, since it begins with a `#`. In fact, gnuplot ignores everything after a `#`. In order to plot the second column as a function of the first, type the command:

```
gnuplot> plot "data" using 1:2 with points
```

The name of the file is within double quotes. After the keyword `using`, we instruct gnuplot which columns to use as the $x$ and $y$ coordinates, respectively. The keywords `with points` instructs gnuplot to add each pair $(x_i, y_i)$ to the plot with points.

The command

```
gnuplot> plot "data" using 1:3 with lines
```

plots the third column as a function of the first, and the keywords `with lines` instruct gnuplot to connect each pair $(x_i, y_i)$ with a straight line segment.

We can combine several plots together in one plot:

```
gnuplot> plot   "data" using 1:3 with points, exp(-0.5*x)
gnuplot> replot "data" using 1:2
gnuplot> replot 2*x
```

The first line plots the 1st and 3rd columns in the file `data` together with the function $\mathrm{e}^{-x/2}$. The second line *adds* the plot of the 1st and 2nd columns in the file `data` and the third line adds the plot of the function $2x$.

There are many powerful ways to use the keyword `using`. Instead of column numbers, we can put mathematical expressions enclosed inside brackets, like `using (...):(...)`. Gnuplot evaluates each expression within the brackets and plots the result. In these expressions, the values

of each column in the file `data` are represented as in the `awk` language. `$i` are variables that expand to the number read from columns `i=1,2,3,...`. Here are some examples:

```
gnuplot> plot "data" using 1:($2*sin($1)*$3) with points
gnuplot> replot 2*x*sin(x)*exp(-x/2)
```

The first line plots the 1st column of the file `data` together with the value $y_i sin(x_i) z_i$, where $y_i$, $x_i$ and $z_i$ are the numbers in the 2nd, 1st and 3rd columns respectively. The second line adds the plot of the function $2x\sin(x)\mathrm{e}^{-x/2}$.

```
gnuplot> plot "data" using (log($1)):(log($2**2))
gnuplot> replot 2*x+log(4)
```

The first line plots the logarithm of the 1st column together with the logarithm of the square of the 2nd column.

We can plot the data written to the standard output of *any* command. Assume that there is a program called `area` that prints the perimeter and area of a circle to the `stdout` in the form shown below:

```
> ./area
R=     3.280000       area=     33.79851
R=     6.280000       area=     123.8994
R=     5.280000       area=     87.58257
R=     4.280000       area=     57.54895
```

The interesting data is at the second and fourth columns. These can be plotted directly with the gnuplot command:

```
gnuplot> plot "< ./area" using 2:4
```

All we have to do is to type the full command after the < within the double quotes. We can create complicated filters using pipes as in the following example:

```
gnuplot> plot \
 "< ./area|sort -g -k 2|awk '{print log($2),log($4)}'" \
 using 1:2
```

The filter produces data to the `stdout`, by combining the action of the commands `area`, `sort` and `awk`. The data printed by the last program is

in two columns and we plot the results `using 1:2`.

In order to save plots in files, we have to change the *terminal* that gnuplot outputs the plots. Gnuplot can produce plots in several languages (e.g. PDF, postscript, SVG, LaTeX, jpeg, png, gif, etc), which can be interpreted and rendered by external programs. By redirecting the output to a file, we can save the plot to the hard disk. For example:

```
gnuplot> plot "data" using 1:3
gnuplot> set terminal jpeg
gnuplot> set output "data.jpg"
gnuplot> replot
gnuplot> set output
gnuplot> set terminal wxt
```

The first line makes the plot as usual. The second one sets the output to be in the JPEG format and the third one sets the name of the file to which the plot will be saved. The fourth lines repeats all the previous plotting commands and the fifth one closes the file `data.jpg`. The last line chooses the interactive terminal `wxt` to be the output of the next plot. High quality images are usually saved in the PDF, encapsulated postcript or SVG format. Use `set terminal pdf,postscript eps` or `svg`, respectively.

And now a few words for 3-dimensional (3d) plotting. The next example uses the command `splot` in order to make a 3d plot of the function $f(x, y) = \mathrm{e}^{-x^2-y^2}$. After you make the plot, you can use the mouse in order to rotate it and view it from a different perspective:

```
gnuplot> set pm3d
gnuplot> set hidden3d
gnuplot> set size ratio 1
gnuplot> set isosamples 50
gnuplot> splot [-2:2][-2:2] exp(-x**2-y**2)
```

If you have data in the form $(x_i, y_i, z_i)$ and you want to create a plot of $z_i = f(x_i, y_i)$, write the data in a file, like in the following example:

```
-1 -1 2.000
-1  0 1.000
-1  1 2.000

 0 -1 1.000
 0  0 0.000
 0  1 1.000
```

```
 1 -1  2.000
 1  0  1.000
 1  1  2.000
```

Note the empty line that follows the change of the value of the first column. If the name of the file is `data3`, then you can plot the data with the commands:

```
gnuplot> set pm3d
gnuplot> set hidden3d
gnuplot> set size ratio 1
gnuplot> splot "data3" with lines
```

We close this section with a few words on parametric plots. A parametric plot on the plane (2-dimensions) is a curve $(x(t), y(t))$, where $t$ is a parameter. A parametric plot in space (3-dimensions) is a surface $(x(u,v)\ , y(u,v), z(u,v))$, where $(u,v)$ are parameters. The following commands plot the circle $(\sin t, \cos t)$ and the sphere $(\cos u \cos v,\ \cos u \sin v,\ \sin u)$:

```
gnuplot> set parametric
gnuplot> plot sin(t),cos(t)
gnuplot> splot cos(u)*cos(v),cos(u)*sin(v),sin(u)
```

## 1.6 Shell Scripting

Complicated system administration tasks are not among the strengths of the Fortran programming language. But in a typical GNU/Linux environment, there exist many powerful tools that can be used very effectively for this purpose. This way, one can use Fortran for the high performance and scientific computing part of the project and leave the administration and trivial data analysis tasks to other, external, programs.

One can avoid repeating the same sequence of commands by coding them in a file. An example can be found in the file `script01.csh`:

```
#!/bin/tcsh -f
gfortran area_01.f90 -o area
./area
gfortran area_02.f90 -o area
./area
```

```
gfortran area_03.f90 -o area
./area
gfortran area_04.f90 -o area
./area
```

This is a very simple shell script. The first line instructs the operating system that the lines that follow are to be *interpreted* by the program `/bin/tcsh`[46]. This can be any program in the system, which in our case is the `tcsh` shell. The following lines are valid commands for the shell, one in each line. They compile the Fortran programs found in the files that we created in section 1.4 with `gfortran`, and then they run the executable `./area`. In order to execute the commands in the file, we have to make sure that the file has the appropriate execute permissions. If not, we have to give the command:

```
> chmod u+x script01.csh
```

Then we simply type the path to the file `script01.csh`

```
> ./script01.csh
```

and the above commands are run the one after the other. Some of the versions of the programs that we wrote are asking for input from the `stdin`, which, normally, you have to type on the terminal. Instead of interacting directly with the program, we can write the input data to a file `Input`, and run the command

```
./area < Input
```

A more convenient solution is to use the, so called, "Here Document". A "Here Document" is a section of the script that is treated as if it were a separate file. As such, it can be used as input to programs by sending its "contents" to the `stdin` of the command that runs the program[47]. The "Here Document" does not appear in the filesystem and we don't need to administer it as a regular file. An example of using a "Here Document" can be found in the file `script02.csh`:

```
#!/bin/tcsh -f
```

[46]Use `#!/bin/bash` if you prefer the `bash` shell.

[47]Their great advantage is that we can use variable and command substitution in them, therefore sending this information to the program that we want to run.

```
gfortran area_04.f90 -o area
./area <<EOF
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
10.0
EOF
```

The `stdin` of the command `./area` is redirected to the contents between the lines

```
./area <<EOF
...
EOF
```

The string `EOF` marks the beginning and the end of the "Here Document", and can be any string you like. The last `EOF` has to be placed exactly in the beginning of the line.

The power of shell scripting lies in its programming capabilities: Variables, arrays, loops and conditionals can be used in order to create a complicated program. Shell variables can be used as discussed in section 1.1.2: The value of a variable `name` is `$name` and it can be set with the command `set name = value`. An array is defined, for example, by the command

```
set R = (1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0)
```

and its data can be accessed using the syntax `$R[1] ... $R[10]`.

Lets take a look at the following script:

```
#!/bin/tcsh -f

set files = (area_01.f90 area_02.f90 area_03.f90 area_04.f90)
set R     = (1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0)

echo "Hello $USER Today is " `date`
foreach file ($files)
 echo "# ------------ Working on file $file "
```

```
 gfortran $file -o area
 ./area <<EOF
$R[1]
$R[2]
$R[3]
$R[4]
$R[5]
$R[6]
$R[7]
$R[8]
$R[9]
$R[10]
EOF
 echo "# ------------- Done "
 if( -f AREA.DAT ) cat AREA.DAT
end
```

The first two lines of the script define the values of the arrays `files` (4 values) and `R` (10 values). The command `echo` echoes its argument to the `stdin`. `$USER` is the name of the user running the script. `` `date` `` is an example of *command substitution*: When a command is enclosed between backquotes and is part of a string, then the command is executed and its `stdout` is pasted back to the string. In the example shown above, `` `date` `` is replaced by the current date and time in the format produced by the `date` command.

The `foreach` loop

```
foreach file ($files)
 ...
end
```

is executed once for each of the 4 values of the array `files`. Each time the value of the variable `file` is set equal to one of the values `area_01.f90`, `area_02.f90`, `area_03.f90`, `area_04.f90`. These values can be used by the commands in the loop. Therefore, the command `gfortran $file -o area` compiles a different file each time that it is executed by the loop.

The last line in the loop

```
 if( -f AREA.DAT ) cat AREA.DAT
```

is a conditional. It executes the command `cat AREA.DAT` if the condition `-f AREA.DAT` is true. In this case, `-f` constructs a logical expression which is true when the file `AREA.DAT` exists.

We close this section by presenting a more complicated and advanced

script. It only serves as a demonstration of the shell scripting capabilities. For more information, the reader is referred to the bibliography [13, 14, 15, 16, 17]. Read carefully the commands, as well as the comments which follow the # mark. Then, write the commands to a file `script04.csh`[48], make it an executable file with the command `chmod u+x script04.csh` and give the command

```
> ./script04.csh This is my first serious tcsh script
```

The script will run with the words "This is my first serious tcsh script" as its arguments. Notice how these arguments are manipulated by the script. Then, the script asks for the values of the radii of ten or more circles interactively, so that it will compute their perimeter and area. Type them on the terminal and then observe the script's output, so that you understand the function of each command. You will not regret the time investment!

```
#!/bin/tcsh -f
# Run this script as:
# ./script04.csh Hello this is a tcsh script
#-----------------------------------------------------------------------
# `command` is command substitution: it is replaced by stdout of command
set now = `date` ; set mypc = `uname -a`
# Print information: variables are expanded within double quotes
echo "I am user $user working on the computer $HOST" #HOST is predefined
echo "Today the date is       :  $now"               #now  is defined above
echo "My home directory is    :  $home"              #home is predefined
echo "My current directory is:  $cwd"                #cwd changes with cd
echo "My computer runs        :  $mypc"              #mypc is defined above
echo "My process id is        :  $$   "              #$$   is predefined
# Manipulate the command line: ($#argv is number of elements in array argv)
echo "The command line has $#argv arguments"
echo "The name of the command I am running is: $0"
echo "Arguments 3rd to last of the command   : $argv[3-]"    #third to last
echo "The last argument is                   : $argv[$#argv]" #last element
echo "All arguments                          : $argv"

# Ask user for input: enter radii of circles
echo -n "Enter radii of circles: " # variable $< stores one line of input
set  Rs = ($<)  #Rs is now an array with all words entered by user
if($#Rs < 10 )then #make a test, need at least 10 of them
 echo "Need more than 10 radii. Exiting...."
 exit(1)
endif
echo "You entered $#Rs radii, the first is $Rs[1] and the last $Rs[$#Rs]"
echo "Rs= $Rs"
# Now, compute the perimeter of each circle:
foreach R ($Rs)
 # -v rad=$R set the awk variable rad equal to $R. pi=atan2(0,-1)=3.14...
 set l = `awk -v rad=$R 'BEGIN{print 2*atan2(0,-1)*rad}'`
```

[48] You will find it also in the accompanying software

```
 echo "Circle with R= $R has perimeter $l"
end
# alias defines a command to do what you want: use awk as a calculator
alias acalc  'awk "BEGIN{print \!* }"' # \!* substitutes args of acalc
echo "Using acalc to compute       2+3=" `acalc 2+3`
echo "Using acalc to compute cos(2*pi)=" `acalc cos(2*atan2(0,-1))`
# Now do the same loop over radii as above in a different way
# while( expression ) is executed as long as "expression" is true
while($#Rs > 0) #executed as long as $Rs contains radii
 set R = $Rs[1] #take first element of $Rs
 shift Rs       #now $Rs has one less element:old $Rs[1] has vanished
 set a = `acalc atan2(0,-1)*${R}*${R}` # =pi*R*R calculated by acalc
 # construct a filename to save the result from the value of R:
 set file = area${R}.dat
 echo "Circle with R= $R has area $a" > $file #save result in a file
end             #end while
# Now look for our files: save their names in an array files:
set files = (`ls -1 area*.dat`)
if( $#files == 0) echo "Sorry, no area files found"
echo "-----------------------------------------------"
echo "files: $files"
ls -l $files
echo "-----------------------------------------------"
echo "And the results for the area are:"
foreach f ($files)
 echo -n "file ${f}: "
 cat $f
end
# now play a little bit with file names:
echo "-----------------------------------------------"
set f = $files[1] # test permissions on first file
# -f, -r, -w, -x, -d test existence of file, rwxd permissions
# the ! negates the expression (true -> false, false -> true)
echo "testing permissions on files:"
if(  -f $f     ) echo "$file exists"
if(  -r $f     ) echo "$file is readable by me"
if(  -w $f     ) echo "$file is writable by be"
if(! -w /bin/ls) echo "/bin/ls is NOT writable by me"
if(! -x $f     ) echo "$file is NOT an executable"
if(  -x /bin/ls) echo "/bin/ls is executable by me"
if(! -d $f     ) echo "$file is NOT a directory"
if(  -d /bin   ) echo "/bin is a directory"
echo "-----------------------------------------------"
# transform the name of a file
set f = $cwd/$f       # add the full path in $f
set filename  = $f:r  # removes extension .dat
set extension = $f:e  # gets    extension .dat
set fdir      = $f:h  # gets    directory of $f
set base      = `basename $f` # removes directory name
echo "file      is: $f"
echo "filename  is: $filename"
echo "extension is: $extension"
echo "directory is: $fdir"
echo "basename  is: $base"
# now transform the name to one with different extension:
set newfile = ${filename}.jpg
echo "jpeg name is: $newfile"
echo "jpeg base is:" `basename $newfile`
if($newfile:e == jpg)echo `basename $newfile` " is a picture"
echo "-----------------------------------------------"
# Now save all data in a file using a "here document"
# A here document starts with <<EOF and ends with a line
```

```
# starting exactly with EOF (EOF can be any string as below)
# In a "here document" we can use variables and command
# substitution:
cat <<AREAS >> areas.dat
# This file contains the areas of circle of given radii
# Computation done by ${user} on ${HOST}. Today is `date`
`cat $files`
AREAS
# now see what we got:
if( -f areas.dat) cat areas.dat
# You can use a "here document" as standard input to any command:
# use gnuplot to save a plot: gnuplot does the job and exits...
gnuplot <<GNU
set terminal jpeg
set output   "areas.jpg"
plot "areas.dat" using 4:7 title "areas.dat",\
     pi*x*x                title "pi*R^2"
set output
GNU
# check our results: display the jpeg file using eog
if( -f areas.jpg) eog areas.jpg &
```

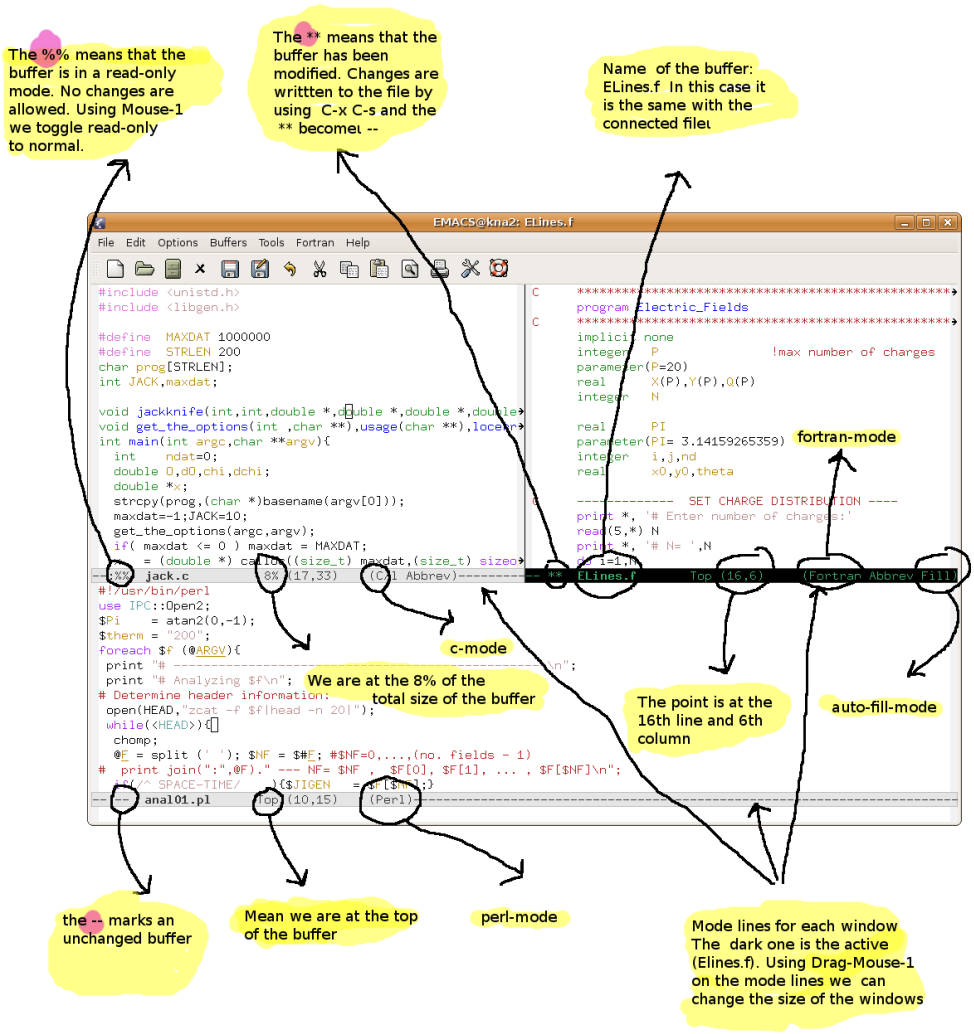



Figure 1.5: In this figure, the Emacs window has been split in three *windows*. The splitting was done horizontally first (`C-x 2`), and then vertically (`C-x 3`). By dragging the mouse (`Drag-Mouse-1`) on the horizontal *mode lines* and vertical lines that separate the windows, we can change window sizes. Notice the useful information diplayed on the mode lines. Each window has one *point* and the *cursor* is on the active window (in this case the window of the buffer named `ELines.f`). A buffer with no active changes in its contents is marked by a `--`, an edited buffer is marked by `**` and a buffer in *read only* mode with (%%). With a mouse click on a %%, we can change them to `--` (so that we can edit) and vice versa. With `Mouse-3` on the name of a *mode* we can activate a choice of *minor modes*. With `Mouse-1` on the name of a mode we ca have access to commands relevant to the mode. The numbers (17,31), (16,6) και (10,15) on the mode lines show the (line,column) of the point location on the respective windows.

| | |
|---|---|
| `awk` | search for and process patterns in a file, |
| `cat` | display, or join, files |
| `cd` | change working directory |
| `chmod` | change the access mode of a file |
| `cp` | copy files |
| `date` | display current time and date |
| `df` | display the amount of available disk space |
| `diff` | display the differences between two files |
| `du` | display information on disk usage |
| `echo` | echo a text string to output |
| `find` | find files |
| `grep` | search for a pattern in files |
| `gzip` | compress files in the gzip (.gz) format (gunzip to uncompress) |
| `head` | display the first few lines of a file |
| `kill` | send a signal (like KILL) to a process |
| `locate` | search for files stored on the system (faster than find) |
| `less` | display a file one screen at a time |
| `ln` | create a link to a file |
| `lpr` | print files |
| `ls` | list information about files |
| `man` | search information about command in man pages |
| `mkdir` | create a directory |
| `mv` | move and/or rename a file |
| `ps` | report information on the processes run on the system |
| `pwd` | print the working directory |
| `rm` | remove (delete) files |
| `rmdir` | remove (delete) a directory |
| `sort` | sort and/or merge files |
| `tail` | display the last few lines of a file |
| `tar` | store or retrieve files from an archive file |
| `top` | dynamic real-time view of processes |
| `wc` | counts lines, words and characters in a file |
| `whatis` | list man page entries for a command |
| `where` | show where a command is located in the path (alternatively: whereis) |
| `which` | locate an executable program using "path" |
| `zip` | create compressed archive in the zip format (.zip) |
| `unzip` | get/list contents of zip archive |

Table 1.1: Basic Unix commands.

Table 1.2: Some intrinsic functions in Fortran.

| Function | Description |
|---|---|
| ABS | modulus of a complex number, absolute value of number |
| ACOS | arccosine of a number |
| ADJUSTL | moves non blank characters of a string to the left |
| ADJUSTR | moves non blank characters of a string to the right |
| AIMAG | imaginary part of a complex number |
| AINT | truncates fractional part but preserves data type |
| ANINT | rounds to nearest whole number but preserves data type |
| ASIN | arcsine of a number |
| ATAN | arctangent of a number |
| ATAN2 | arctangent of arg1 divided by arg2 resolved into the correct quadrant |
| CMPLX | converts to the COMPLEX data type arg1 + i arg2 |
| CONJG | complex conjugate of a complex number |
| COS | cosine of an angle in radians |
| COSH | hyperbolic cosine |
| DATE_AND_TIME | returns current date and time |
| DBLE | converts to the `real(8)` data type |
| DIM | if arg1 > arg2, then returns arg1 - arg2; otherwise 0 |
| DPROD | double precision product of two single precision numbers |
| EXP | exponential |
| EPSILON | Returns a positive number that is negligible compared to 1.0 |
| HUGE | Returns the largest number of the same kind as the argument |
| INT | converts to the INTEGER data type by truncation |
| KIND | Returns the KIND value of argument |
| LEN | Returns the length of a string |

Continued...

Table 1.2: Continued...

| Function | Description |
|---|---|
| LEN_TRIM | returns the length of a string without trailing blanks |
| LGE,LGT,LLE,LLT | string comparison functions |
| LOG | natural logarithm |
| LOG10 | common logarithm |
| MAX | maximum value of arguments |
| MAXEXPONENT | returns the maximum exponent of the same kind as the argument |
| MIN | minimum value of arguments |
| MINEXPONENT | returns the minimum exponent of the same kind as the argument |
| MOD | arg1 modulo arg2 |
| NINT | converts to the INTEGER data type by rounding |
| RANDOM_NUMBER | returns pseudo-random numbers $0 \leq r < 1$ |
| RANDOM_SEED | starts random number generator or returns generator parameters |
| PRECISION | returns the decimal precision of the same kind as the argument |
| REAL | real part of a complex number |
| REAL | converts to the REAL data type |
| SIGN | if arg2 $< 0$, then returns -arg1; else +arg1 |
| SIN | sine of an angle in radians |
| SINH | hyperbolic sine |
| SQRT | square root |
| TAN | tangent of an angle in radians |
| TANH | hyperbolic tangent |
| TINY | returns the smallest positive number of the same kind as the argument |
| TRIM | returns string with trailing blanks removed |
| Array functions | |
| ALL | true if all values are true |
| ALLOCATED | array allocation status |
| ANY | true if any values are true |
| COUNT | number of elements in an array |

Continued...

Table 1.2: Continued...

| Function | Description |
|---|---|
| DOT_PRODUCT | dot product of two rank-one arrays |
| LBOUND | lower dimension bounds of an array |
| MATMUL | matrix multiplication |
| MAXLOC | location of a maximum value in an array |
| MAXVAL | maximum value in an array |
| MERGE | merge arrays under mask |
| MINLOC | location of a minimum value in an array |
| MINVAL | minimum value in an array |
| PACK | pack an array into an array of rank one under a mask |
| PRODUCT | product of array elements |
| RESHAPE | reshape an array |
| SHAPE | shape of an array or scalar |
| SIZE | size of an array |
| SPREAD | replicate an array by adding a dimension |
| SUM | sum of array elements |
| TRANSPOSE | transpose an array of rank two |
| UBOUND | upper dimension bounds of an array |
| UNPACK | unpack an array of rank one into an array under a mask |

Table 1.3: Basic Emacs commands.

| | | |
|---|---|---|
| **Leaving Emacs** | | |
| suspend Emacs (or iconify it under X) | `C-z` | |
| exit Emacs permanently | `C-x C-c` | |
| **Files** | | |
| **read** a file into Emacs | `C-x C-f` | |
| **save** a file back to disk | `C-x C-s` | |
| save **all** files | `C-x s` | |
| **insert** contents of another file into this buffer | `C-x i` | |
| toggle read-only status of buffer | `C-x C-q` | |
| **Getting Help** | | |
| The help system is simple. Type `C-h` (or `F1`) and follow the directions. If you are a first-time user, type `C-h t` for a **tutorial**. | | |
| remove help window | `C-x 1` | |
| apropos: show commands matching a string | `C-h a` | |
| describe the function a key runs | `C-h k` | |
| describe a function | `C-h f` | |
| get mode-specific information | `C-h m` | |
| **Error Recovery** | | |
| **abort** partially typed or executing command | `C-g` | |
| **recover** files lost by a system crash | `M-x recover-session` | |
| **undo** an unwanted change | `C-x u, C-_` or `C-/` | |
| restore a buffer to its original contents | `M-x revert-buffer` | |
| redraw garbaged screen | `C-l` | |
| **Incremental Search** | | |
| search forward | `C-s` | |
| search backward | `C-r` | |
| regular expression search | `C-M-s` | |
| abort current search | `C-g` | |
| Use `C-s` or `C-r` again to repeat the search in either direction. If Emacs is still searching, `C-g` cancels only the part not matched. | | |
| **Motion** | | |
| **entity to move over** | **backward** | **forward** |
| character | `C-b` | `C-f` |
| word | `M-b` | `M-f` |
| line | `C-p` | `C-n` |
| go to line beginning (or end) | `C-a` | `C-e` |
| go to buffer beginning (or end) | `M-<` | `M->` |
| scroll to next screen | `C-v` | |

Continued...

Table 1.3: Continued...

| | | |
|---|---|---|
| scroll to previous screen | M-v | |
| scroll left | C-x < | |
| scroll right | C-x > | |
| scroll current line to center of screen | C-u C-l | |
| **Killing and Deleting** | | |
| **entity to kill** | **backward** | **forward** |
| character (delete, not kill) | DEL | C-d |
| word | M-DEL | M-d |
| line (to end of) | M-0 C-k | C-k |
| kill **region** | C-w | |
| copy region to kill ring | M-w | |
| yank back last thing killed | C-y | |
| replace last yank with previous kill | M-y | |
| **Marking** | | |
| set mark here | C-@ or C-SPC | |
| exchange point and mark | C-x C-x | |
| mark **paragraph** | M-h | |
| mark entire **buffer** | C-x h | |
| **Query Replace** | | |
| interactively replace a text string | M-% *or* M-x query-replace | |
| using regular expressions | M-x query-replace-regexp | |
| **Buffers** | | |
| select another buffer | C-x b | |
| list all buffers | C-x C-b | |
| kill a buffer | C-x k | |
| **Multiple Windows** | | |
| When two commands are shown, the second is a similar command for a frame instead of a window. | | |
| delete all other windows | C-x 1 | C-x 5 1 |
| split window, above and below | C-x 2 | C-x 5 2 |
| delete this window | C-x 0 | C-x 5 0 |
| split window, side by side | C-x 3 | |
| switch cursor to another window | C-x o | C-x 5 o |
| grow window taller | C-x ^ | |
| shrink window narrower | C-x { | |
| grow window wider | C-x } | |

Continued...

Table 1.3: Continued...

| | |
|---|---|
| **Formatting** | |
| indent current **line** (indent code etc) | `TAB` |
| insert newline after point | `C-o` |
| fill paragraph | `M-q` |
| **Case Change** | |
| uppercase word | `M-u` |
| lowercase word | `M-l` |
| capitalize word | `M-c` |
| uppercase region | `C-x C-u` |
| lowercase region | `C-x C-l` |
| **The Minibuffer** | |
| The following keys are defined in the minibuffer. | |
| complete as much as possible | `TAB` |
| complete up to one word | `SPC` |
| complete and execute | `RET` |
| **abort command** | `C-g` |
| Type `C-x ESC ESC` to edit and repeat the last command that used the minibuffer. Type `F10` to activate menu bar items on text terminals. | |
| **Spelling Check** | |
| check spelling of current word | `M-$` |
| check spelling of all words in region | `M-x ispell-region` |
| check spelling of entire buffer | `M-x ispell-buffer` |
| On the fly spell checking | `M-x flyspell-mode` |
| **Info – Getting Help Within Emacs** | |
| enter the Info documentation reader | `C-h i` |
| scroll forward | `SPC` |
| scroll reverse | `DEL` |
| **next** node | `n` |
| **previous** node | `p` |
| move **up** | `u` |
| select menu item by name | `m` |
| return to last node you saw | `l` |
| return to directory node | `d` |
| go to top node of Info file | `t` |
| go to any node by name | `g` |
| **quit** Info | `q` |

# Chapter 2

# Kinematics

In this chapter we show how to program simple kinematic equations of motion of a particle and how to do basic analysis of numerical results. We use simple methods for plotting and animating trajectories on the two dimensional plane and three dimensional space. In section 2.3 we study numerical errors in the calculation of trajectories of freely moving particles bouncing off hard walls and obstacles. This will be a prelude to the study of the integration of the *dynamical* equations of motion that we will introduce in the following chapters.

## 2.1 Motion on the Plane

When a particle moves on the plane, its position can be given in Cartesian coordinates $(x(t), y(t))$. These, as a function of time, describe the particle's trajectory. The position vector is $\vec{r}(t) = x(t)\,\hat{x} + y(y)\,\hat{y}$, where $\hat{x}$ and $\hat{y}$ are the unit vectors on the $x$ and $y$ axes respectively. The velocity vector is $\vec{v}(t) = v_x(t)\,\hat{x} + v_y(t)\,\hat{y}$ where

$$\begin{aligned} \vec{v}(t) &= \frac{d\vec{r}(t)}{dt} \\ v_x(t) = \frac{dx(t)}{dt} \qquad & v_y(t) = \frac{dy(t)}{dt}\,, \end{aligned} \tag{2.1}$$

The acceleration $\vec{a}(t) = a_x(t)\,\hat{x} + a_y(t)\,\hat{y}$ is given by

$$\begin{aligned} \vec{a}(t) = \frac{d\vec{v}(t)}{dt} &= \frac{d^2\vec{r}(t)}{dt^2} \\ a_x(t) = \frac{dv_x(t)}{dt} = \frac{d^2x(t)}{dt^2} \qquad & a_y(t) = \frac{dv_y(t)}{dt} = \frac{d^2y(t)}{dt^2}\,. \end{aligned} \tag{2.2}$$

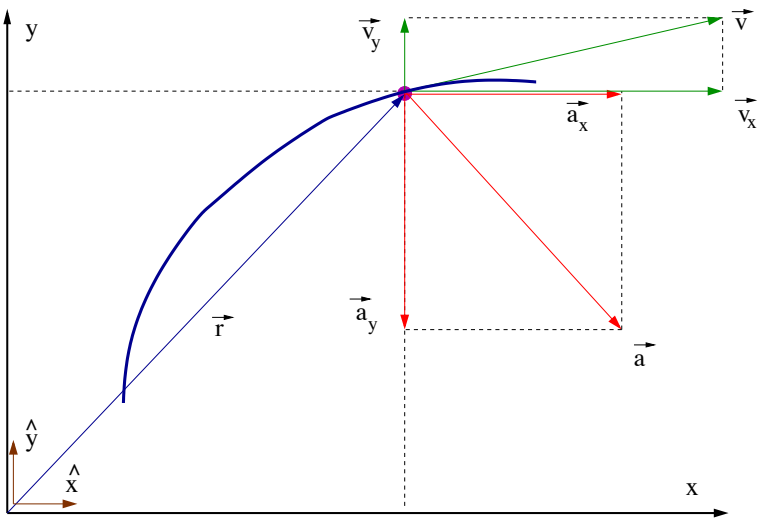



Figure 2.1: The trajectory of a particle moving in the plane. The figure shows its position vector $\vec{r}$, velocity $\vec{v}$ and acceleration $\vec{a}$ and their Cartesian components in the chosen coordinate system at a point of the trajectory.

In this section we study the kinematics of a particle trajectory, therefore we assume that the functions $(x(t), y(t))$ are known. By taking their derivatives, we can compute the velocity and the acceleration of the particle in motion. We will write simple programs that compute the values of these functions in a time interval $[t_0, t_f]$, where $t_0$ is the initial and $t_f$ is the final time. The continuous functions $x(t), y(t), v_x(t), v_y(t)$ are approximated by a discrete sequence of their values at the times $t_0, t_0 + \delta t, t_0 + 2\delta t, t_0 + 3\delta t, \ldots$ such that $t_0 + n\delta t \leq t_f$.

We will start the design of our program by forming a generic template to be used in all of the problems of interest. Then we can study each problem of particle motion by programming only the equations of motion without worrying about the less important tasks, like input/output, user interface etc. Figure 2.2 shows a *flowchart* of the basic steps in the algorithm. The first part of the program declares variables and defines the values of the fixed parameters (like $\pi = 3.1459\ldots$, $g = 9.81$, etc). The program starts by interacting with the user ("user interface") and asks for the values of the variables $x_0$, $y_0$, $t_0$, $t_f$, $\delta t \ldots$. The program prints these values to the `stdout` so that the user can check them for correctness and store them in her data.

The main calculation is performed in a *loop* executed while $t \leq t_f$. The values of the positions and the velocities $x(t), y(t), v_x(t), v_y(t)$ are

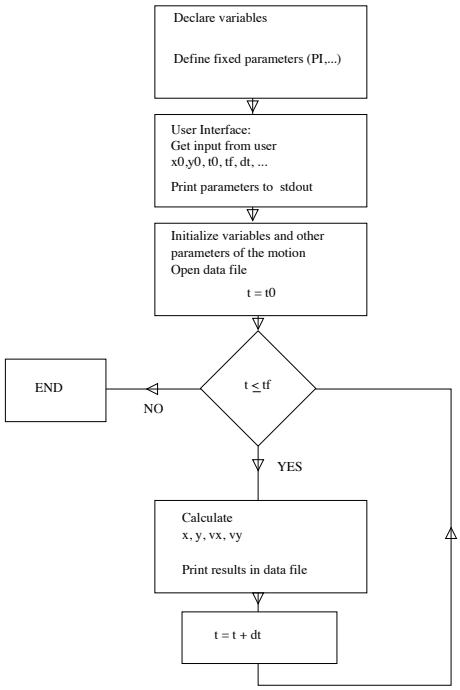



Figure 2.2: The flowchart of a typical program computing the trajectory of a particle from its (kinematic) equations of motion.

calculated and printed in a file together with the time $t$. At this point we fix the format of the program output, something that is very important to do it in a consistent and convenient way for easing data analysis. We choose to print the values t, x, y, vx, vy in five columns in each line of the output file.

The specific problem that we are going to solve is the computation of the trajectory of the circular motion of a particle on a circle with center $(x_0, y_0)$ and radius $R$ with constant angular velocity $\omega$. The position on the circle can be defined by the angle $\theta$, as can be seen in figure 2.3. We define the initial position of the particle at time $t_0$ to be $\theta(t_0) = 0$.

The equations giving the position of the particle at time $t$ are

$$\begin{aligned} x(t) &= x_0 + R\cos\left(\omega(t - t_0)\right) \\ y(t) &= y_0 + R\sin\left(\omega(t - t_0)\right) \ . \end{aligned} \tag{2.3}$$

Taking the derivative w.r.t. $t$ we obtain the velocity

$$\begin{aligned} v_x(t) &= -\omega R\sin\left(\omega(t - t_0)\right) \\ v_y(t) &= \omega R\cos\left(\omega(t - t_0)\right) \ , \end{aligned} \tag{2.4}$$

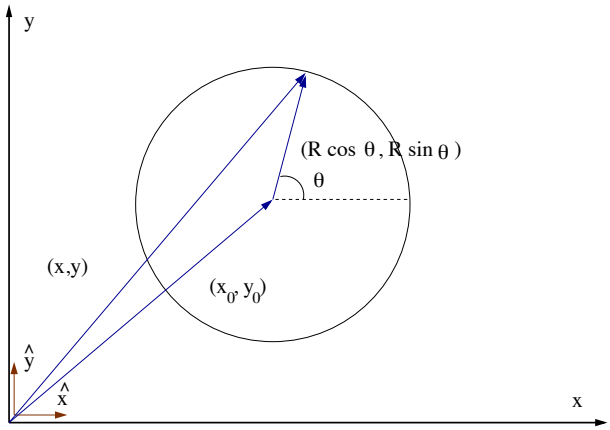



Figure 2.3: The trajectory of a particle moving on a circle with constant angular velocity calculated by the program `Circle.f90`.

and the acceleration

$$\begin{aligned} a_x(t) &= -\omega^2 R\cos\left(\omega(t-t_0)\right) = -\omega^2(x(t)-x_0) \\ a_y(t) &= -\omega^2 R\sin\left(\omega(t-t_0)\right) = -\omega^2(y(t)-y_0)\,. \end{aligned} \tag{2.5}$$

We note that the above equations imply that $\vec{R}\cdot\vec{v}=0$ ($\vec{R}\equiv\vec{r}-\vec{r}_0$, $\vec{v}\perp\vec{R}$, $\vec{v}$ tangent to the trajectory) and $\vec{a}=-\omega^2\vec{R}$ ($\vec{R}$ and $\vec{a}$ anti-parallel, $\vec{a}\perp\vec{v}$).

The data structure is quite simple. The constant angular velocity $\omega$ is stored in the `REAL` variable `omega`. The center of the circle $(x_0, y_0)$, the radius $R$ of the circle and the angle $\theta$ are stored in the `REAL` variables `x0, y0, R, theta`. The times at which we calculate the particle's position and velocity are defined by the parameters $t_0, t_f, \delta t$ and are stored in the `REAL` variables `t0, tf, dt`. The current position $(x(t), y(t))$ is calculated and stored in the `REAL` variables `x, y` and the velocity $(v_x(t), v_y(t))$ in the `REAL` variables `vx, vy`. The declarations of the variables are put in the beginning of the program:

```
real :: x0,y0,R,x,y,vx,vy,t,t0,tf,dt
real :: theta,omega
real, parameter :: PI=3.1415927
```

were we defined the value[4] of $\pi = 3.1415927$ by using the `parameter` specification.

[4]The reader is reminded that `REAL` variables are stored in 4 bytes and have an accuracy of about 7 decimal digits.

The user interface of the program is the interaction of the program with the user and, in our case, it is the part of the program where the user enters the parameters `omega, x0, y0, R, t0, tf, dt`. The program issues a prompt with the names the variables expected to be read. This is done using simple `print` statements. The variables are read from the `stdin` by simple `read` statements and the values entered by the user are printed to the `stdout`[2]:

```
print *,'# Enter omega:'
read  *,omega
print *,'# Enter center of circle (x0,y0) and radius R:'
read  *,x0,y0,R
print *,'# Enter t0,tf,dt:'
read  *,t0,tf,dt
print *,'# omega= ',omega
print *,'# x0= ',x0,' y0= ',y0,' R= ',R
print *,'# t0= ',t0,' tf= ',tf,' dt= ',dt
```

Next, the program initializes the state of the computation. This includes checking the validity of the parameters entered by the user, so that the computation will be possible. For example, the program computes the expression `2.0*PI/omega`, where it is assumed that `omega` has a non zero value. We will also demand that $R > 0$ and $\omega > 0$. An `if` statement will make those checks and if the parameters have illegal values, the `stop` statement will stop the program execution[3]. The program opens the file `Circle.dat` for writing the calculated values of the position and the velocity of the particle.

```
if(R     .le. 0.0) stop 'Illegal value of R'
if(omega .le. 0.0) stop 'Illegal value of omega'
print *,'# T= ',2.0*PI/omega
open(unit=11,file='Circle.dat')
```

If $R \leq 0$ or $\omega \leq 0$ the corresponding `stop` statements are executed which end the program execution. The optional error messages are included after the `stop` statements which are printed to the `stdout`. The value of the period $T = 2\pi/\omega$ is also calculated and printed for reference.

The `open` statement uses *unit* `11` for writing to the file `Circle.dat`.

[2]This is done so that the used can check for typos and see the actual value read by the program. By redirecting the `stdout` of a file on the hard disk, the parameters can be saved for future reference and used in data analysis.

[3]Note that there are more assumptions that need to be checked by the program. We leave this as an exercise for the reader.

The choice of the unit number is free for the programmer to choose. We recommend using the units 10 to 99 for input/output to files[4].

The main calculation is performed within the loop

```
t   =  t0
do while(t .le. tf)
 .........
 t  =  t + dt
enddo
```

The first statement sets the initial value of the time. The statements between the `do while(`*condition*`)` and `enddo` are executed as long as *condition* has a `.TRUE.` value. The statement `t=t+dt` increments the time and this is necessary in order not to enter into an infinite loop. The commands put in place of the dots ..... calculate the position and the velocity and print them to the file `Circle.dat`:

```
 theta = omega * (t-t0)
 x  =  x0+R*cos(theta)
 y  =  y0+R*sin(theta)
 vx =  -omega*R*sin(theta)
 vy =   omega*R*cos(theta)
 write(11,*)t,x,y,vx,vy
```

Notice the use of the intrinsic functions `sin` and `cos` that calculate the sine and cosine of an angle expressed in radians. We use the intermediate variable `theta` in order to store the phase $\theta(t) = \omega(t - t_0)$. The command `write(11,*)` writes the variables `t,x,y,vx,vy` to the unit 11, which has been associated to the file `Circle.dat` with the open statement shown above.

The program is stored in the file `Circle.f90` and can be found in the accompanied software. The *extension* `.f90` is used by the compiler in order to denote source code written in free format Fortran language. Compilation and running can be done using the commands:

```
> gfortran Circle.f90 -o cl
> ./cl
```

The switch `-o cl` forces the compiler `gfortran` to write the binary com-

[4]Some numbers can be reserved for special files, like 5 for `stdin`, 6 for `stdout` and 0 for `stderr`. Using numbers larger than 99 can lead to portability problems.

mands executed by the program to the file[5] `cl`. The command `./cl` loads the program instructions to the computer memory for execution. When the programs starts execution, it first asks for the parameter data and then performs the calculation. A typical session looks like:

```
> gfortran Circle.f90 -o cl
> ./cl
 # Enter omega:
1.0
 # Enter center of circle (x0,y0) and radius R:
1.0 1.0 0.5
 # Enter t0,tf,dt:
0.0 20.0 0.01
 # omega=   1.
 # x0=   1. y0=   1. R=   0.5
 # t0=   0. tf=   20. dt=   0.00999999978
 # T=   6.28318548
```

The lines shown above that start with a # character are printed by the program and lines without # are the values of the parameters entered interactively by the user. The user types in the parameters and then presses the `Enter` key in order for the program to read them. Here we have $\omega = 1.0$, $x_0 = y_0 = 1.0$, $R = 0.5$, $t_0 = 0.0$, $t_f = 20.0$ and $\delta t = 0.01$.

You can execute the above program many times for different values of the parameter by writing the parameter values in a file using an editor. For example, in the file `Circle.in` type the following data:

```
1.0              omega
1.0  1.0 0.5     (x0, y0) , R
0.0 20.0 0.01    t0 tf dt
```

Each line has the parameters read by the program with a read statement (a *record*). The rest of the line is ignored by the program and the user can write anything she likes as a comment on how to use the parameters. The program can read the above values of the parameters with the command:

```
> ./cl < Circle.in > Circle.out
```

The command `./cl` runs the commands found in the executable file `./cl`. The `< Circle.in` redirects the contents of the file `Circle.in` to the standard input (`stdin`) of the command `./cl`. This way the program reads

[5] If omitted, the executable file has the default name `a.out`.

in the values of the parameters from the contents of the file `Circle.in`. The `> Circle.out` redirects the standard output (`stdout`) of the command `./cl` to the file `Circle.out`. Its contents can be inspected after the execution of the program with the command `cat`:

```
> cat Circle.out
 # Enter omega:
 # Enter center of circle (x0,y0) and radius R:
 # Enter t0,tf,dt:
 # omega=    1.
 # x0=    1. y0=    1. R=    0.5
 # t0=    0. tf=    20. dt=    0.00999999978
 # T=    6.28318548
```

We list the full program in `Circle.f90` below:

```
!==========================================================
!File Circle.f90
!Constant angular velocity circular motion
!Set (x0,y0) center of circle, its radius R and omega.
!At t=t0, the particle is at theta=0
!----------------------------------------------------------
program Circle
 implicit none
!----------------------------------------------------------
!Declaration of variables
 real :: x0,y0,R,x,y,vx,vy,t,t0,tf,dt
 real :: theta,omega
 real, parameter :: PI=3.1415927
!----------------------------------------------------------
!Ask user for input:
 print *,'# Enter omega:'
 read  *,omega
 print *,'# Enter center of circle (x0,y0) and radius R:'
 read  *,x0,y0,R
 print *,'# Enter t0,tf,dt:'
 read  *,t0,tf,dt
 print *,'# omega= ',omega
 print *,'# x0= ',x0,' y0= ',y0,' R= ',R
 print *,'# t0= ',t0,' tf= ',tf,' dt= ',dt
!----------------------------------------------------------
!Initialize
 if(R     .le. 0.0) stop 'Illegal value of R'
 if(omega .le. 0.0) stop 'Illegal value of omega'
 print *,'# T= ',2.0*PI/omega
 open(unit=11,file='Circle.dat')
!----------------------------------------------------------
```

```
!Compute:
 t    =  t0
 do while(t .le. tf)
  theta = omega * (t-t0)
  x   =  x0+R*cos(theta)
  y   =  y0+R*sin(theta)
  vx  =  -omega*R*sin(theta)
  vy  =   omega*R*cos(theta)
  write(11,*)t,x,y,vx,vy
  t   =  t + dt
 enddo
 close(11)
end program Circle
```

### 2.1.1 Plotting Data

We use `gnuplot` for plotting the data produced by our programs. The file `Circle.dat` has the time `t` and the components `x, y, vx, vy` in five columns. Therefore we can plot the functions $x(t)$ and $y(t)$ by using the `gnuplot` commands:

```
gnuplot> plot   "Circle.dat" using 1:2 with lines title "x(t)"
gnuplot> replot "Circle.dat" using 1:3 with lines title "y(t)"
```

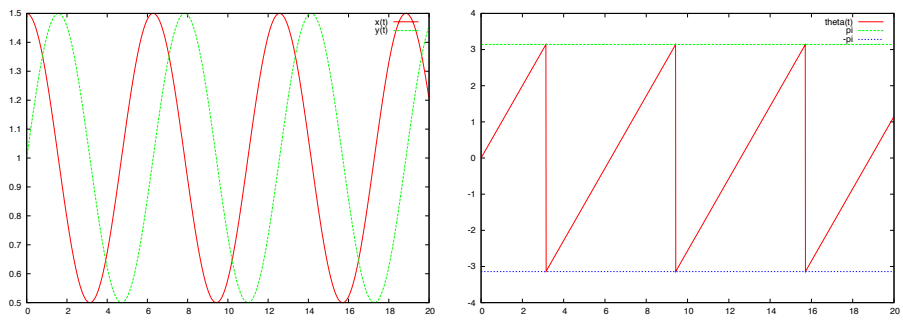



Figure 2.4: The plots $(x(t), y(t))$ (left) and $\theta(t)$ (right) from the data in `Circle.dat` for $\omega = 1.0$, $x_0 = y_0 = 1.0$, $R = 0.5$, $t_0 = 0.0$, $t_f = 20.0$ and $\delta t = 0.01$.

The second line puts the second plot together with the first one. The results can be seen in figure 2.4.

Let's see now how we can make the plot of the function $\theta(t)$. We can do that using the raw data from the file `Circle.dat` within `gnuplot`, without having to write a new program. Note that $\theta(t) = \tan^{-1}\left((y - y_0)/(x - x_0)\right)$.

The function atan2 is available in gnuplot[6] as well as in Fortran. Use the online help system in gnuplot in order to see its usage:

```
gnuplot> help atan2
 The `atan2(y,x)` function returns the arc tangent (inverse
 tangent) of the ratio of the real parts of its arguments.
 `atan2` returns its argument in radians or degrees, as
 selected by `set angles`, in the correct quadrant.
```

Therefore, the right way to call the function is atan2(y-y0,x-x0). In our case x0=y0=1 and x, y are in the 2nd and 3rd columns of the file Circle.dat. We can construct an *expression* after the using command as in page 60, where $2 is the value of the second and $3 the value of the third column:

```
gnuplot> x0 = 1 ; y0 = 1
gnuplot> plot "Circle.dat" using 1:(atan2($3-y0,$2-x0)) \
                           with lines title "theta(t)",pi,-pi
```

The second command is broken in two lines by using the character \ so that it fits conveniently in the text[7]. Note how we defined the values of the variables x0, y0 and how we used them in the expression atan2($3-x0,$2-y0). We also plot the lines which graph the constant functions $f_1(t) = \pi$ and $f_2(t) = -\pi$ which mark the limit values of $\theta(t)$. The gnuplot variable[8] pi is predefined and can be used in formed expressions. The result can be seen in the left plot of figure 2.4.

The velocity components $(v_x(t), v_y(t))$ as function of time as well as the trajectory $\vec{r}(t)$ can be plotted with the commands:

```
gnuplot> plot   "Circle.dat" using 1:4 title "v_x(t)" \
                             with lines
gnuplot> replot "Circle.dat" using 1:5 title "v_y(t)" \
                             with lines
gnuplot> plot   "Circle.dat" using 2:3 title "x-y"
                             with lines
```

We close this section by showing how to do a simple animation of the particle trajectory using gnuplot. There is a file animate2D.gnu in the accompanied software which you can copy in the directory where you

[6]The command help functions will show you all the available functions in gnuplot.

[7]This can be done on the gnuplot command line as well.

[8]Use the command show variables in order to see the current/default values of gnuplot variables.

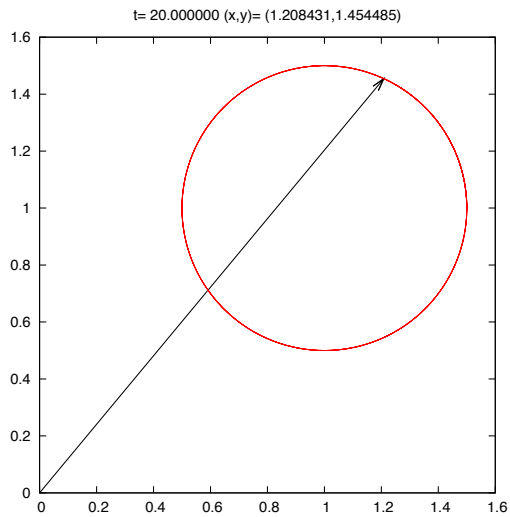



Figure 2.5: The particle trajectory plotted by the `gnuplot` program in the file `animate2D.gnu` of the accompanied software. The position vector is shown at a given time `t`, which is marked on the title of the plot together with the coordinates `(x,y)`. The data is produced by the program `Circle.f90` described in the text.

have the data file `Circle.dat`. We are not going to explain how it works[9] but how to use it in order to make your own animations. The final result is shown in figure 2.5. All that you need to do is to define the data file[10], the initial time `t0`, the final time `tf` and the time step `dt`. These times can be different from the ones we used to create the data in `Circle.dat`. A full animation session can be launched using the commands:

```
gnuplot> file = "Circle.dat"
gnuplot> set xrange [0:1.6]; set yrange [0:1.6]
gnuplot> t0   = 0; tf = 20 ; dt = 0.1
gnuplot> load "animate2D.gnu"
```

The first line defines the data file that `animate2D.gnu` reads data from. The second line sets the range of the plots and the third line defines the time parameters used in the animation. The final line launches the animation. If you want to rerun the animation, you can repeat the last two commands as many times as you want using the same or different parameters. E.g. if you wish to run the animation at "half the speed"

[9]You are most welcome to study the commands in the script and guess how it works of course!

[10]It can be *any* file that has $(t, x, y)$ in the 1st, 2nd and 3rd columns respectively.

you should simply redefine `dt=0.05` and set the initial time to `t0=0`:

```
gnuplot> t0   = 0; dt = 0.05
gnuplot> load "animate2D.gnu"
```

### 2.1.2 More Examples

We are now going to apply the steps described in the previous section to other examples of motion on the plane. The first problem that we are going to discuss is that of the small oscillations of a simple pendulum. Figure 2.6 shows the single oscillating degree of freedom $\theta(t)$, which is the small angle that the pendulum forms with the vertical direction. The motion is periodic with angular frequency $\omega = \sqrt{g/l}$ and period

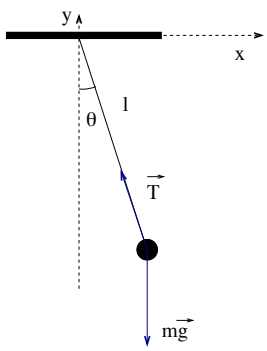



Figure 2.6: The simple pendulum whose motion for $\theta \ll 1$ is described by the program `SimplePendulum.f90`.

$T = 2\pi/\omega$. The angular velocity is computed from $\dot{\theta} \equiv d\theta/dt$ which gives

$$\begin{aligned} \theta(t) &= \theta_0 \cos\left(\omega(t - t_0)\right) \\ \dot{\theta}(t) &= -\omega\theta_0 \sin\left(\omega(t - t_0)\right) \end{aligned} \tag{2.6}$$

We have chosen the initial conditions $\theta(t_0) = \theta_0$ and $\dot{\theta}(t_0) = 0$. In order to write the equations of motion in the Cartesian coordinate system shown in figure 2.6 we use the relations

$$\begin{aligned} x(t) &= l \sin\left(\theta(t)\right) \\ y(t) &= -l \cos\left(\theta(t)\right) \\ v_x(t) &= \frac{dx(t)}{dt} = l\dot{\theta}(t) \cos\left(\theta(t)\right) \\ v_y(t) &= \frac{dy(t)}{dt} = l\dot{\theta}(t) \sin\left(\theta(t)\right) \ . \end{aligned} \tag{2.7}$$

These are similar to the equations (2.3) and (2.4) that we used in the case of the circular motion of the previous section. Therefore the structure of the program is quite similar. Its final form, which can be found in the file `SimplePendulum.f90`, is:

```
!=============================================================
!File SimplePendulum.f90
!Set pendulum original position at theta0 with no initial speed
!-------------------------------------------------------------
program SimplePendulum
 implicit none
!-------------------------------------------------------------
!Declaration of variables
 real :: l,x,y,vx,vy,t,t0,tf,dt
 real :: theta,theta0,dtheta_dt,omega
 real, parameter ::PI=3.1415927,g=9.81
!-------------------------------------------------------------
!Ask user for input:
 print *,'# Enter l: '
 read  *,l
 print *,'# Enter theta0:'
 read  *,theta0
 print *,'# Enter t0,tf,dt:'
 read  *,t0,tf,dt
 print *,'# l=  ',l ,' theta0= ',theta0
 print *,'# t0= ',t0,' tf= ',tf,' dt= ',dt
!-------------------------------------------------------------
!Initialize
 omega = sqrt(g/l)
 print *,'# omega= ',omega,' T= ',2.0*PI/omega
 open(unit=11,file='SimplePendulum.dat')
!-------------------------------------------------------------
!Compute:
 t   =  t0
 do while(t .le. tf)
  theta     =        theta0*cos(omega*(t-t0))
  dtheta_dt = -omega*theta0*sin(omega*(t-t0))
  x  =  l*sin(theta)
  y  = -l*cos(theta)
  vx =  l*dtheta_dt*cos(theta)
  vy =  l*dtheta_dt*sin(theta)
  write(11,100)t,x,y,vx,vy,theta,dtheta_dt
  t  =  t + dt
 enddo
 close(11)
100 FORMAT(7G15.7)
end program SimplePendulum
```

We note that the acceleration of gravity $g$ is hard coded in the program and that the user can only set the length $l$ of the pendulum. The data file `SimplePendulum.dat` produced by the program, contains two extra columns with the current values of $\theta(t)$ and the angular velocity $\dot{\theta}(t)$. The statement `write(11,100)` writes to the unit 11 according to the format set by the `FORMAT` statement, found in the line labeled by the label 100. This is done so that we can be sure that the data is printed in one line for each value of $t$ (see the discussion on page 48).

A simple session for the study of the above problem is shown below[11]:

```
> gfortran SimplePendulum.f90 -o sp
> ./sp
 # Enter l:
1.0
 # Enter theta0:
0.314
 # Enter t0,tf,dt:
0 20 0.01
 # l=        1.        theta0=    0.31400001
 # t0=       0.        tf=        20. dt=    0.00999999978
 # omega=    3.132092 T=          2.0060668
> gnuplot
gnuplot> plot   "SimplePendulum.dat" u 1:2 w l t "x(t)"
gnuplot> plot   "SimplePendulum.dat" u 1:3 w l t "y(t)"
gnuplot> plot   "SimplePendulum.dat" u 1:4 w l t "v_x(t)"
gnuplot> replot "SimplePendulum.dat" u 1:5 w l t "v_y(t)"
gnuplot> plot   "SimplePendulum.dat" u 1:6 w l t "theta(t)"
gnuplot> replot "SimplePendulum.dat" u 1:7 w l t "theta'(t)"
gnuplot> plot   [-0.6:0.6][-1.1:0.1] "SimplePendulum.dat" \
                                     u 2:3 w l t "x-y"
gnuplot> file = "SimplePendulum.dat"
gnuplot> t0=0;tf=20.0;dt=0.1
gnuplot> set xrange  [-0.6:0.6];set yrange [-1.1:0.1]
gnuplot> load "animate2D.gnu"
```

The next example is the study of the trajectory of a particle shot near the earth's surface[12] when we consider the effect of air resistance to be negligible. Then, the equations describing the trajectory of the particle

[11]Notice that we replaced the command "`using 1:2 with lines title`" with "`u 1:2 w lines t`". These abbreviations can be done with every `gnuplot` command if an abbreviation uniquely determines a command.

[12]I.e. $\vec{g} =$ const. and the Coriolis force can be ignored.

and its velocity are given by the parametric equations

$$\begin{aligned} x(t) &= v_{0x}t \\ y(t) &= v_{0y}t - \frac{1}{2}gt^2 \\ v_x(t) &= v_{0x} \\ v_y(t) &= v_{0y} - gt\,, \end{aligned} \tag{2.8}$$

where $t$ is the parameter. The initial conditions are $x(0) = y(0) = 0$, $v_x(0) = v_{0x} = v_0 \cos\theta$ and $v_y(0) = v_{0y} = v_0 \sin\theta$, as shown in figure 2.7.

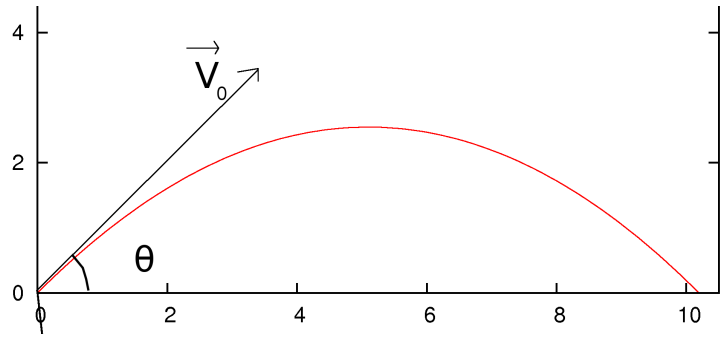



Figure 2.7: The trajectory of a particle moving under the influence of a constant gravitational field. The initial conditions are set to $x(0) = y(0) = 0$, $v_x(0) = v_{0x} = v_0 \cos\theta$ and $v_y(0) = v_{0y} = v_0 \sin\theta$.

The structure of the program is similar to the previous ones. The user enters the magnitude of the particle's initial velocity and the shooting angle $\theta$ in *degrees*. The initial time is taken to be $t_0 = 0$. The program calculates $v_{0x}$ and $v_{0y}$ and prints them to the `stdout`. The data is written to the file `Projectile.dat`. The full program is listed below and it can be found in the file `Projectile.f90` in the accompanied software:

```
!==========================================================
!File Projectile.f90
!Shooting a projectile near the earth surface.
!No air resistance.
!Starts at (0,0), set (v0,theta).
!----------------------------------------------------------
program Projectile
 implicit none
!----------------------------------------------------------
!Declaration of variables
 real :: x0,y0,R,x,y,vx,vy,t,tf,dt
```

```
 real :: theta,v0x,v0y,v0
 real, parameter :: PI=3.1415927,g=9.81
!-----------------------------------------------------------------
!Ask user for input:
 print *,'# Enter v0,theta (in degrees):'
 read  *,v0,theta
 print *,'# Enter tf,dt:'
 read  *, tf,dt
 print *,'# v0= ',v0,' theta= ',theta,'o (degrees)'
 print *,'# t0= ',0.0,' tf= ',tf,' dt= ',dt
!-----------------------------------------------------------------
!Initialize
 if( v0    .le. 0.0) stop 'Illegal value of v0<=0'
 if( theta .le. 0.0 .or. theta .ge. 90.0) &
      stop 'Illegal value of theta'
 theta  = (PI/180.0)*theta !convert to radians
 v0x    = v0*cos(theta)
 v0y    = v0*sin(theta)
 print *,'# v0x = ',v0x,' v0y= ',v0y
 open(unit=11,file='Projectile.dat')
!-----------------------------------------------------------------
!Compute:
 t   =  0.0
 do while(t .le. tf)
  x  =  v0x * t
  y  =  v0y * t - 0.5*g*t*t
  vx =  v0x
  vy =  v0y     -     g*t
  write(11,*)t,x,y,vx,vy
  t  =  t + dt
 enddo
 close(11)
end program Projectile
```

A typical session for the study of this problem is shown below:

```
> gfortran Projectile.f90 -o pj
> ./pj
 # Enter v0,theta (in degrees):
10 45
 # Enter tf,dt:
1.4416 0.001
 # v0=    10.0000000   theta= 45.000000  o (degrees)
 # t0=     0.0000000   tf=     1.4416000 dt=   1.00000005E-03
 # v0x =   7.0710678   v0y=    7.0710678
> gnuplot
gnuplot> plot   "Projectile.dat" using 1:2 w l t "x(t)"
gnuplot> replot "Projectile.dat" using 1:3 w l t "y(t)"
```

```
gnuplot> plot   "Projectile.dat" using 1:4 w l t "v_x(t)"
gnuplot> replot "Projectile.dat" using 1:5 w l t "v_y(t)"
gnuplot> plot   "Projectile.dat" using 2:3 w l t "x-y"
gnuplot> file = "Projectile.dat"
gnuplot> set xrange [0:10.3];set yrange [0:10.3]
gnuplot> t0=0;tf=1.4416;dt=0.05
gnuplot> load "animate2D.gnu"
```

Next, we will study the effect of air resistance of the form $\vec{F} = -mk\vec{v}$. The solutions to the equations of motion

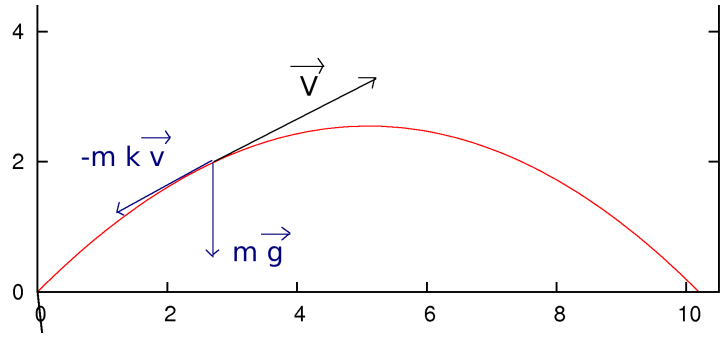



Figure 2.8: The forces that act on the particle of figure 2.7 when we assume air resistance of the form $\vec{F} = -mk\vec{v}$.

$$
\begin{aligned}
a_x &= \frac{dv_x}{dt} = -kv_x \\
a_y &= \frac{dv_y}{dt} = -kv_y - g
\end{aligned}
\tag{2.9}
$$

with initial conditions $x(0) = y(0) = 0$, $v_x(0) = v_{0x} = v_0 \cos\theta$ and $v_y(0) = v_{0y} = v_0 \sin\theta$ are[13]

$$
\begin{aligned}
v_x(t) &= v_{0x} \mathrm{e}^{-kt} \\
v_y(t) &= \left(v_{0y} + \frac{g}{k}\right) \mathrm{e}^{-kt} - \frac{g}{k} \\
x(t) &= \frac{v_{0x}}{k}\left(1 - \mathrm{e}^{-kt}\right) \\
y(t) &= \frac{1}{k}\left(v_{0y} + \frac{g}{k}\right)\left(1 - \mathrm{e}^{-kt}\right) - \frac{g}{k}t
\end{aligned}
\tag{2.10}
$$

Programming the above equations is as easy as before, the only difference being that the user needs to provide the value of the constant $k$.

[13]The proof of equations (2.10) is left as an exercise for the reader.

The full program can be found in the file `ProjectileAirResistance.f90` and it is listed below:

```
!==========================================================
!File ProjectileAirResistance.f90
!Shooting a projectile near the earth surface
!with air resistance
!Starts at (0,0), set k, (v0,theta).
!----------------------------------------------------------
program ProjectileAirResistance
 implicit none
!----------------------------------------------------------
!Declaration of variables
 real :: x0,y0,R,x,y,vx,vy,t,tf,dt,k
 real :: theta,v0x,v0y,v0
 real, parameter :: PI=3.1415927,g=9.81
!----------------------------------------------------------
!Ask user for input:
 print *,'# Enter k, v0,theta (in degrees):'
 read  *,k, v0,theta
 print *,'# Enter tf,dt:'
 read  *, tf,dt
 print *,'# k = ',k
 print *,'# v0= ',v0,' theta= ',theta,'o (degrees)'
 print *,'# t0= ',0.0,' tf= ',tf,' dt= ',dt
!----------------------------------------------------------
!Initialize
 if( v0    .le. 0.0) stop 'Illegal value of v0<=0'
 if( k     .le. 0.0) stop 'Illegal value of k <=0'
 if( theta .le. 0.0 .or. theta .ge. 90.0) &
      stop 'Illegal value of theta'
 theta  = (PI/180.0)*theta !convert to radians
 v0x    = v0*cos(theta)
 v0y    = v0*sin(theta)
 print *,'# v0x = ',v0x,' v0y= ',v0y
 open(unit=11,file='ProjectileAirResistance.dat')
!----------------------------------------------------------
!Compute:
 t   =  0.0
 do while(t .le. tf)
  x  =  (v0x/k)*(1.0-exp(-k*t))
  y  =  (1.0/k)*(v0y+(g/k))*(1.0-exp(-k*t))-(g/k)*t
  vx =  v0x*exp(-k*t)
  vy =  (v0y+(g/k))*exp(-k*t)-(g/k)
  write(11,*)t,x,y,vx,vy
  t  =  t + dt
 enddo
 close(11)
```

```
end program ProjectileAirResistance
```

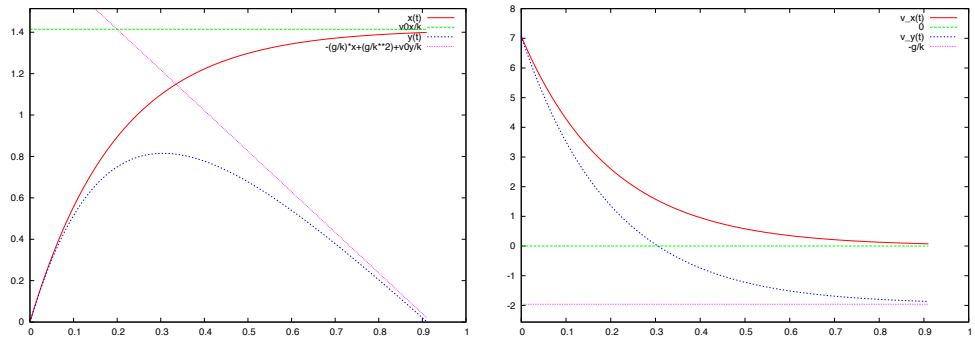



Figure 2.9: The plots of $x(t)$,$y(t)$ (left) and $v_x(t)$,$v_y(t)$ (right) from the data produced by the program `ProjectileAirResistance.f90` for $k = 5.0$, $v_0 = 10.0$, $\theta = \pi/4$, $t_f = 0.91$ and $\delta t = 0.001$. We also plot the asymptotes of these functions as $t \to \infty$.

We also list the commands of a typical session of the study of the problem:

```
> gfortran ProjectileAirResistance.f90 -o pja
> ./pja
 # Enter k, v0,theta (in degrees):
5.0 10.0 45
 # Enter tf,dt:
0.91 0.001
 # k =   5.
 # v0=   10. theta=   45.o (degrees)
 # t0=   0. tf=   0.910000026 dt=   0.00100000005
 # v0x =   7.07106781 v0y=   7.07106781
> gnuplot
gnuplot> v0x = 10*cos(pi/4) ; v0y = 10*sin(pi/4)
gnuplot> g = 9.81 ; k = 5
gnuplot> plot [:][:v0x/k+0.1]  "ProjectileAirResistance.dat" \
         using 1:2 with lines title "x(t)",v0x/k
gnuplot> replot                "ProjectileAirResistance.dat" \
         using 1:3 with lines title "y(t)",\
         -(g/k)*x+(g/k**2)+v0y/k
gnuplot> plot [:][-g/k-0.6:]   "ProjectileAirResistance.dat" \
         using 1:4 with lines title "v_x(t)",0
gnuplot> replot                "ProjectileAirResistance.dat" \
         using 1:5 with lines title "v_y(t)",-g/k
gnuplot> plot                  "ProjectileAirResistance.dat" \
         using 2:3 with lines title "With air resistance k=5.0"
gnuplot> replot                "Projectile.dat"              \
```

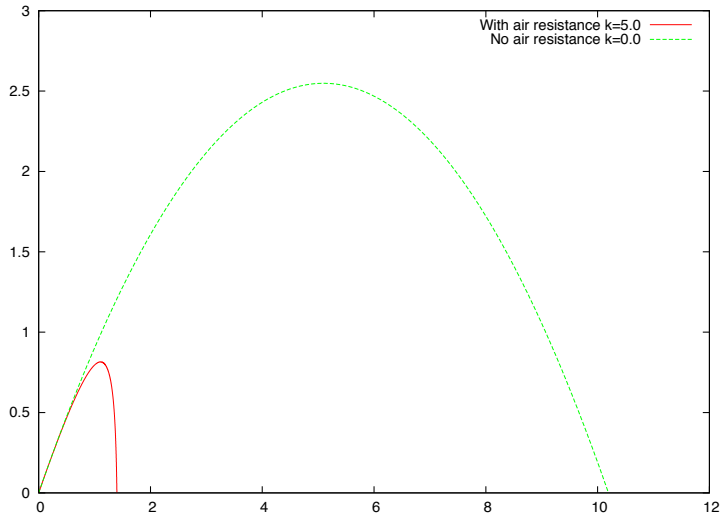



Figure 2.10: Trajectories of the particles shot with $v_0 = 10.0$, $\theta = \pi/4$ in the absence of air resistance and when the air resistance is present in the form $\vec{F} = -mk\vec{v}$ with $k = 5.0$.

```
         using 2:3 with lines title "No air resistance k=0.0"
gnuplot> file = "ProjectileAirResistance.dat"
gnuplot> set xrange [0:1.4];set yrange [0:1.4]
gnuplot> t0=0;tf=0.91;dt=0.01
gnuplot> load "animate2D.gnu"
```

Long commands have been continued to the next line as before. We defined the `gnuplot` variables `v0x`, `v0y`, `g` and `k` to have the values that we used when running the program. We can use them in order to construct the asymptotes of the plotted functions of time. The results are shown in figures 2.9 and 2.10.

The last example of this section will be that of the anisotropic harmonic oscillator. The force on the particle is

$$F_x = -m\omega_1^2 x \qquad F_y = -m\omega_2^2 y \tag{2.11}$$

where the "spring constants" $k_1 = m\omega_1^2$ and $k_2 = m\omega_2^2$ are different in the directions of the axes $x$ and $y$. The solutions of the dynamical equations of motion for $x(0) = A$, $y(0) = 0$, $v_x(0) = 0$ and $v_y(0) = \omega_2 A$ are

$$\begin{aligned} x(t) &= A\cos(\omega_1 t) \qquad y(t) = A\sin(\omega_2 t) \\ v_x(t) &= -\omega_1 A\sin(\omega_1 t) \qquad v_y(t) = \omega_2 A\cos(\omega_2 t)\,. \end{aligned} \tag{2.12}$$

If the angular frequencies $\omega_1$ and $\omega_2$ satisfy certain relations, the trajectories of the particle are closed and self intersect at a given number of

points. The proof of these relations, as well as their numerical confirmation, is left as an exercise for the reader. The program listed below is in the file `Lissajoux.f90`:

```
!==========================================================
!File Lissajous.f90
!Lissajous curves (special case)
!x(t)= cos(o1 t), y(t)= sin(o2 t)
!----------------------------------------------------------
program Lissajous
 implicit none
!----------------------------------------------------------
!Declaration of variables
 real x0,y0,R,x,y,vx,vy,t,t0,tf,dt
 real o1,o2,T1,T2
 real, parameter :: PI=3.1415927
!----------------------------------------------------------
!Ask user for input:
 print *,'# Enter omega1 and omega2:'
 read  *,o1,o2
 print *,'# Enter tf,dt:'
 read  *,tf,dt
 print *,'# o1= ',o1, ' o2= ',o2
 print *,'# t0= ',0.0,' tf= ',tf,' dt= ',dt
!----------------------------------------------------------
!Initialize
 if(o1.le.0.0 .or. o2.le.0.0) stop 'omega1 or omega2<=0'
 T1 = 2.0*PI/o1
 T2 = 2.0*PI/o2
 print *,'# T1= ',T1,' T2= ',T2
 open(unit=11,file='Lissajous.dat')
!----------------------------------------------------------
!Compute:
 t   =  0.0
 do while(t .le. tf)
  x  =  cos(o1*t)
  y  =  sin(o2*t)
  vx = -o1*sin(o1*t)
  vy =  o2*cos(o2*t)
  write(11,*)t,x,y,vx,vy
  t  =  t + dt
 enddo
 close(11)
end program Lissajous
```

We have set $A = 1$ in the program above. The user must enter the two angular frequencies $\omega_1$ and $\omega_2$ and the corresponding times. A typical

session for the study of the problem is shown below:

```
> gfortran  Lissajous.f90 -o lsj
> ./lsj
 # Enter omega1 and omega2:
3 5
 # Enter tf,dt:
10.0 0.01
 # o1=   3. o2=   5.
 # t0=   0. tf=   10. dt=   0.00999999978
 # T1=   2.09439516 T2=   1.2566371
>gnuplot
gnuplot> plot   "Lissajous.dat" using 1:2 w l t "x(t)"
gnuplot> replot "Lissajous.dat" using 1:3 w l t "y(t)"
gnuplot> plot   "Lissajous.dat" using 1:4 w l t "v_x(t)"
gnuplot> replot "Lissajous.dat" using 1:5 w l t "v_y(t)"
gnuplot> plot   "Lissajous.dat" using 2:3 w l t "x-y for 3:5"
gnuplot> file = "Lissajous.dat"
gnuplot> set xrange [-1.1:1.1];set yrange [-1.1:1.1]
gnuplot> t0=0;tf=10;dt=0.1
gnuplot> load "animate2D.gnu"
```

The results for $\omega_1 = 3$ and $\omega_2 = 5$ are shown in figure 2.11.

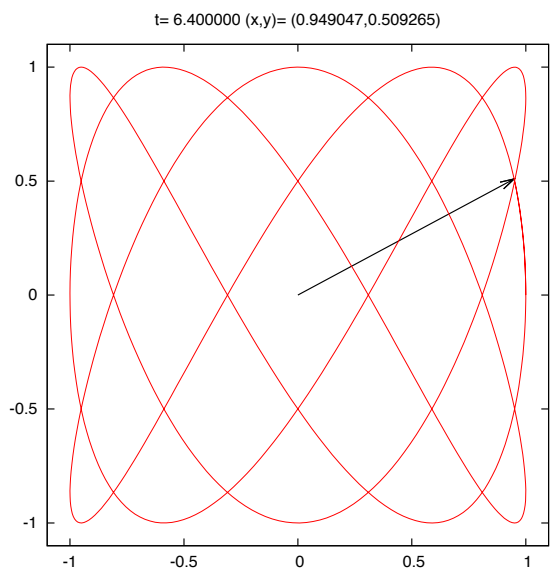



Figure 2.11: The trajectory of the anisotropic oscillator with $\omega_1 = 3$ and $\omega_2 = 5$.

## 2.2 Motion in Space

By slightly generalizing the methods described in the previous section, we will study the motion of a particle in three dimensional space. All we have to do is to add an extra equation for the coordinate $z(t)$ and the component of the velocity $v_z(t)$. The structure of the programs will be exactly the same as before.

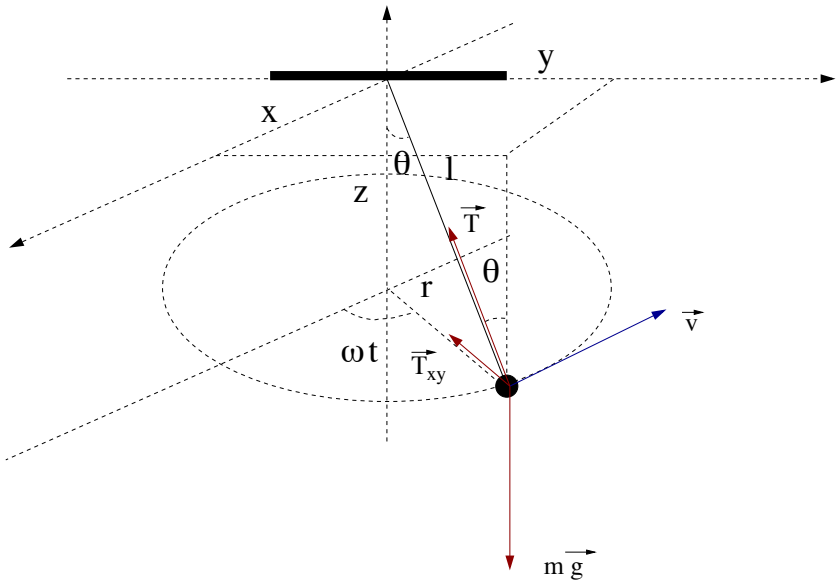



Figure 2.12: The conical pendulum of the program `ConicalPendulum.f90`.

The first example is the conical pendulum, which can be seen in figure 2.12. The particle moves on the $xy$ plane with constant angular velocity $\omega$. The equations of motion are derived from the relations

$$T_z = T\cos\theta = mg \qquad T_{xy} = T\sin\theta = m\omega^2 r\,, \tag{2.13}$$

where $r = l\sin\theta$. Their solution[14] is

$$\begin{aligned} x(t) &= r\cos\omega t \\ y(t) &= r\sin\omega t \\ z(t) &= -l\cos\theta\,, \end{aligned} \tag{2.14}$$

[14] One has to choose appropriate initial condition. Exercise: find them!

where we have to substitute the values

$$\begin{aligned} \cos\theta &= \frac{g}{\omega^2 l} \\ \sin\theta &= \sqrt{1-\cos^2\theta} \\ r &= \frac{g}{\omega^2}\frac{\sin\theta}{\cos\theta}\,. \end{aligned} \tag{2.15}$$

For the velocity components we obtain

$$\begin{aligned} v_x &= -r\omega\sin\omega t \\ v_y &= r\omega\cos\omega t \\ v_z &= 0\,. \end{aligned} \tag{2.16}$$

Therefore we must have

$$\omega \geq \omega_{\text{min}} = \sqrt{\frac{g}{l}}\,, \tag{2.17}$$

and when $\omega \to \infty$, $\theta \to \pi/2$.

In the program that we will write, the user must enter the parameters $l$, $\omega$, the final time $t_f$ and the time step $\delta t$. We take $t_0 = 0$. The convention that we follow for the output of the results is that they should be written in a file where the first 7 columns are the values of $t$, $x$, $y$, $z$, $v_x$, $v_y$ and $v_z$. Each line in this file is long and, in order to prevent Fortran from breaking it into two separate lines, we have to give an explicit format specification. See the discussion on page 48. The full program is listed below:

```
!==============================================================
!File ConicalPendulum.f90
!Set pendulum angular velocity omega and display motion in 3D
!--------------------------------------------------------------
program ConicalPendulum
 implicit none
!--------------------------------------------------------------
!Declaration of variables
 real :: l,r,x,y,z,vx,vy,vz,t,tf,dt
 real :: theta,cos_theta,sin_theta,omega
 real, parameter :: PI=3.1415927,g=9.81
!--------------------------------------------------------------
!Ask user for input:
 print *,'# Enter l,omega: '
 read  *,l,omega
 print *,'# Enter tf,dt:'
```

```
 read  *,tf,dt
 print *,'# l=  ',l            ,' omega=      ',omega
 print *,'# T=  ',2.0*PI/omega,' omega_min= ',sqrt(g/l)
 print *,'# t0= ',0.0,' tf= ',tf,' dt= ',dt
!-----------------------------------------------------------
!Initialize
 cos_theta = g/(omega*omega*l)
 if( cos_theta .ge. 1) stop 'cos(theta)>= 1'
 sin_theta = sqrt(1.0-cos_theta*cos_theta)
 z = -g/(omega*omega) !they remain constant throught
 vz= 0.0              !the  motion
 r =  g/(omega*omega)*sin_theta/cos_theta
 open(unit=11,file='ConicalPendulum.dat')
!-----------------------------------------------------------
!Compute:
 t   = 0.0
 do while(t .le. tf)
  x  =  r*cos(omega*t)
  y  =  r*sin(omega*t)
  vx = -r*sin(omega*t)*omega
  vy =  r*cos(omega*t)*omega
  write(11,100)t,x,y,z,vx,vy,vz
  t  =  t + dt
 enddo
 close(11)
100 FORMAT(20G15.7)
end program ConicalPendulum
```

In order to compile and run the program we can use the commands shown below:

```
> gfortran  ConicalPendulum.f90 -o cpd
> ./cpd
 # Enter l,omega:
1.0 6.28
 # Enter tf,dt:
10.0 0.01
 # l=    1. omega=        6.28000021
 # T=    1.00050724 omega_min=    3.132092
 # t0=   0. tf=   10. dt=   0.00999999978
```

The results are recorded in the file `ConicalPendulum.dat`. In order to plot the functions $x(t)$, $y(t)$, $z(t)$, $v_x(t)$, $v_y(t)$, $v_z(t)$ we give the following `gnuplot` commands:

```
> gnuplot
gnuplot> plot   "ConicalPendulum.dat" u 1:2 w l t "x(t)"
```

```
gnuplot> replot "ConicalPendulum.dat" u 1:3 w l t "y(t)"
gnuplot> replot "ConicalPendulum.dat" u 1:4 w l t "z(t)"
gnuplot> plot   "ConicalPendulum.dat" u 1:5 w l t "v_x(t)"
gnuplot> replot "ConicalPendulum.dat" u 1:6 w l t "v_y(t)"
gnuplot> replot "ConicalPendulum.dat" u 1:7 w l t "v_z(t)"
```

The results are shown in figure 2.13. In order to make a three dimen-

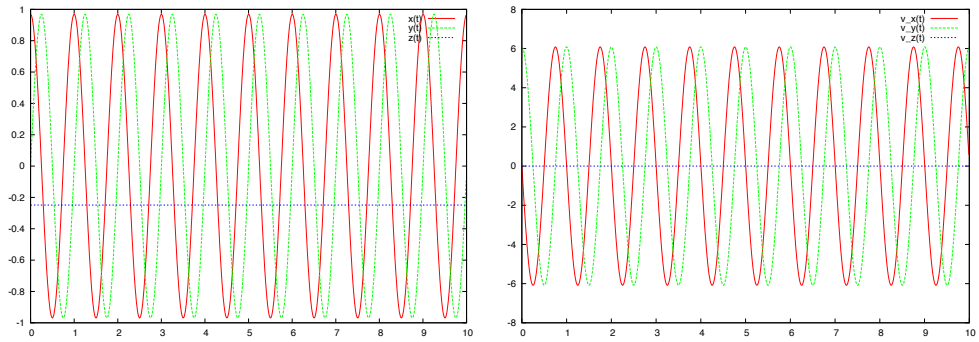



Figure 2.13: The plots of the functions $x(t), y(t), z(t), v_x(t), v_y(t), v_z(t)$ of the program `ConicalPendulum.f90` for $\omega = 6.28$, $l = 1.0$.

sional plot of the trajectory, we should use the `gnuplot` command `splot`:

```
gnuplot> splot "ConicalPendulum.dat" u 2:3:4 w l t "r(t)"
```

The result is shown in figure 2.14. We can click on the trajectory and rotate it and view it from a different angle. We can change the plot limits with the command:

```
gnuplot> splot [-1.1:1.1][-1.1:1.1][-0.3:0.0] \
 "ConicalPendulum.dat" using 2:3:4 w l t "r(t)"
```

We can animate the trajectory of the particle by using the file `animate3D.gnu` from the accompanying software. The commands are similar to the ones we had to give in the two dimensional case for the planar trajectories when we used the file `animate2D.gnu`:

```
gnuplot> set xrange [-1.1:1.1];set yrange [-1.1:1.1]
gnuplot> set zrange [-0.3:0]
gnuplot> t0=0;tf=10;dt=0.1
gnuplot> load "animate3D.gnu"
```

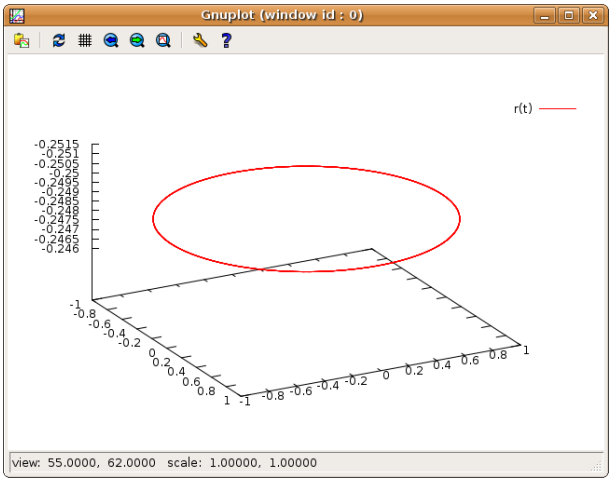



Figure 2.14: The plot of the particle trajectory $\vec{r}(t)$ of the program `ConicalPendulum.f90` for $\omega = 6.28$, $l = 1.0$. We can click and drag with the mouse on the window and rotate the curve and see it from a different angle. At the bottom left of the window, we see the viewing direction, given by the angles $\theta = 55.0$ degrees (angle with the $z$ axis) and $\phi = 62$ degrees (angle with the $x$ axis).

The result can be seen in figure 2.15. The program `animate3D.gnu` can be used on the data file of any program that prints `t x y z` as the first words on each of its lines. All we have to do is to change the value of the `file` variable in gnuplot.

Next, we will study the trajectory of a charged particle in a homogeneous magnetic field $\vec{B} = B\hat{z}$. At time $t_0$, the particle is at $\vec{r}_0 = x_0\hat{x}$ and its velocity is $\vec{v}_0 = v_{0y}\hat{y} + v_{0z}\hat{z}$, see figure 2.16. The magnetic force on the particle is $\vec{F} = q(\vec{v} \times \vec{B}) = qBv_y\hat{x} - qBv_x\hat{y}$ and the equations of motion are

$$\begin{aligned} a_x &= \frac{dv_x}{dt} = \omega v_y \qquad \omega \equiv \frac{qB}{m} \\ a_y &= \frac{dv_y}{dt} = -\omega v_x \\ a_z &= 0\,. \end{aligned} \tag{2.18}$$

By integrating the above equations with the given initial conditions we obtain

$$\begin{aligned} v_x(t) &= v_{0y} \sin \omega t \\ v_y(t) &= v_{0y} \cos \omega t \\ v_z(t) &= v_{0z}\,. \end{aligned} \tag{2.19}$$

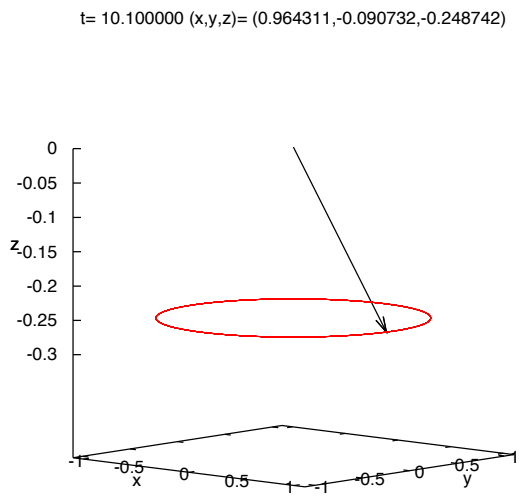



Figure 2.15: The particle trajectory $\vec{r}(t)$ computed by the program `ConicalPendulum.f90` for $\omega = 6.28$, $l = 1.0$ and plotted by the `gnuplot` script `animate3D.gnu`. The title of the plot shows the current time and the particles coordinates.

Integrating once more, we obtain the position of the particle as a function of time

$$\begin{aligned} x(t) &= \left(x_0 + \frac{v_{0y}}{\omega}\right) - \frac{v_{0y}}{\omega}\cos\omega t = x_0 \cos\omega t \\ y(t) &= \frac{v_{0y}}{\omega}\sin\omega t = -x_0 \sin\omega t \quad \text{με} \quad x_0 = -\frac{v_{0y}}{\omega} \\ z(t) &= v_{0z} t\,, \end{aligned} \tag{2.20}$$

where we have chosen $x_0 = -v_{0y}/\omega$. This choice places the center of the circle, which is the projection of the trajectory on the $xy$ plane, to be at the origin of the coordinate system. The trajectory is a helix with radius $R = -x_0$ and pitch $v_{0z}T = 2\pi v_{0z}/\omega$.

We are now ready to write a program that calculates the trajectory given by (2.20). The user enters the parameters $v_0$ and $\theta$, shown in figure 2.16, as well as the angular frequency $\omega$ (Larmor frequency). The components of the initial velocity are $v_{0y} = v_0 \cos\theta$ and $v_{0z} = v_0 \sin\theta$. The initial position is calculated from the equation $x_0 = -v_{0y}/\omega$. The program can be found in the file `ChargeInB.f90`:

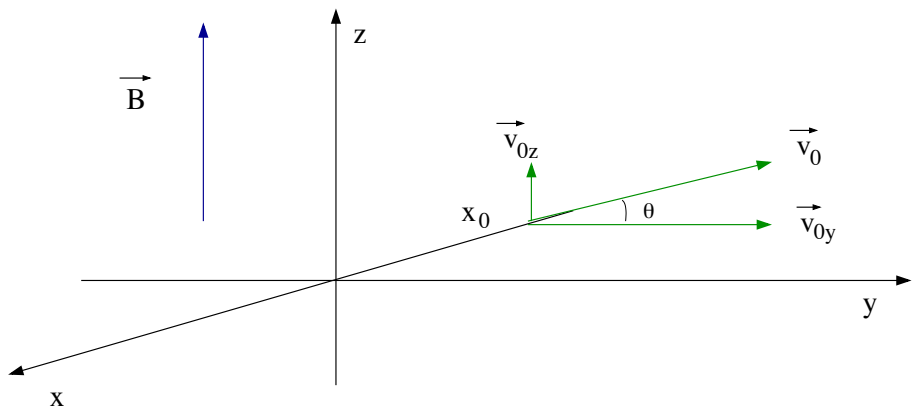



Figure 2.16: A particle at time $t_0 = 0$ is at the position $\vec{r}_0 = x_0\hat{x}$ with velocity $\vec{v}_0 = v_{0y}\hat{y} + v_{0z}\hat{z}$ in a homogeneous magnetic field $\vec{B} = B\hat{z}$.

```
!==========================================================
!File ChargeInB.f90
!A charged particle of mass m and charge q enters a magnetic
!field B in +z direction. It enters with velocity
!v0x=0,v0y=v0 cos(theta),v0z=v0 sin(theta), 0<=theta<pi/2
!at the position x0=-v0y/omega, omega=q B/m
!
!Enter v0 and theta and see trajectory from
!t0=0 to tf at step dt
!----------------------------------------------------------
program ChargeInB
 implicit none
!----------------------------------------------------------
!Declaration of variables
 real :: x,y,z,vx,vy,vz,t,tf,dt
 real :: x0,y0,z0,v0x,v0y,v0z,v0
 real :: theta,omega
 real, parameter :: PI=3.1415927
!----------------------------------------------------------
!Ask user for input:
 print *,'# Enter omega: '
 read  *,omega
 print *,'# Enter v0, theta (degrees):'
 read  *,v0,theta
 print *,'# Enter tf,dt:'
 read  *,tf,dt
 print *,'# omega=  ',omega ,' T=     ',2.0*PI/omega
 print *,'# v0=     ',v0,    ' theta= ',theta,'o (degrees)'
 print *,'# t0=     ',0.0,   ' tf=    ',tf,' dt= ',dt
!----------------------------------------------------------
```

```
!Initialize
 if(theta.lt.0.0 .or. theta.ge.90.0)stop 'Illegal 0<theta<90'
 theta = (PI/180.0)*theta !convert to radians
 v0y   = v0*cos(theta)
 v0z   = v0*sin(theta)
 print *,'# v0x= ',0.0,' v0y= ',v0y,' v0z= ',v0z
 x0    = - v0y/omega
 print *,'# x0=  ',x0, ' y0=  ',0.0,' z0=  ',0.0
 print *,'# xy plane: Circle with center (0,0) and R= ',ABS(x0)
 print *,'# step of helix: s=v0z*T= ',v0z*2.0*PI/omega
 open(unit=11,file='ChargeInB.dat')
!-----------------------------------------------------------------
!Compute:
 t   = 0.0
 vz  = v0z
 do while(t .le. tf)
  x  =  x0*cos(omega*t)
  y  = -x0*sin(omega*t)
  z  =  v0z*t
  vx =  v0y*sin(omega*t)
  vy =  v0y*cos(omega*t)
  write(11,100)t,x,y,z,vx,vy,vz
  t  =  t + dt
 enddo
 close(11)
100 FORMAT(20G15.7)
end program ChargeInB
```

A typical session in which we calculate the trajectories shown in figures 2.17 and 2.18 is shown below:

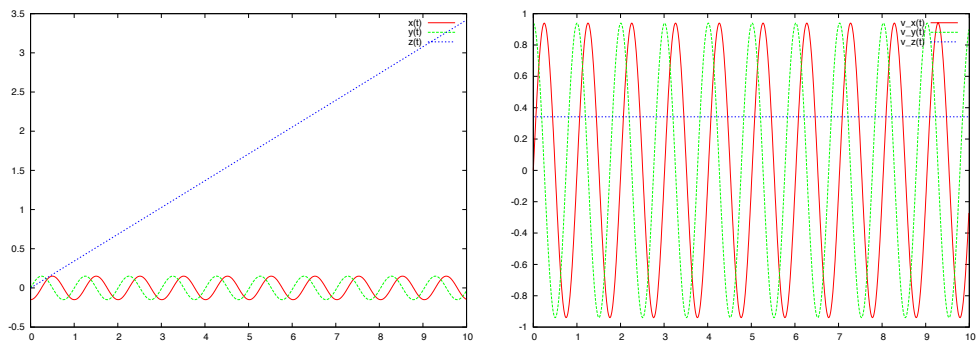



Figure 2.17: The plots of the $x(t), y(t), z(t), v_x(t), v_y(t), v_z(t)$ functions calculated by the program in `ChargeInB.f90` for $\omega = 6.28$, $x_0 = 1.0$, $\theta = 20$ degrees.

```
> gfortran  ChargeInB.f90 -o chg
> ./chg
 # Enter omega:
6.28
 # Enter v0, theta (degrees):
1.0 20
 # Enter tf,dt:
10 0.01
 # omega=    6.28000021 T=       1.00050724
 # v0=       1. theta=   20.o (degrees)
 # t0=       0. tf=      10. dt=   0.00999999978
 # v0x=   0. v0y=   0.939692616 v0z=   0.342020124
 # x0=   -0.149632573 y0=    0. z0=    0.
 # xy plane: Circle with center (0,0) and R=   0.149632573
 # step of helix: s=v0z*T=   0.342193604
> gnuplot
gnuplot> plot   "ChargeInB.dat" u 1:2   w l title "x(t)"
gnuplot> replot "ChargeInB.dat" u 1:3   w l title "y(t)"
gnuplot> replot "ChargeInB.dat" u 1:4   w l title "z(t)"
gnuplot> plot   "ChargeInB.dat" u 1:5   w l title "v_x(t)"
gnuplot> replot "ChargeInB.dat" u 1:6   w l title "v_y(t)"
gnuplot> replot "ChargeInB.dat" u 1:7   w l title "v_z(t)"
gnuplot> splot  "ChargeInB.dat" u 2:3:4 w l title "r(t)"
gnuplot> file = "ChargeInB.dat"
gnuplot> set xrange [-0.65:0.65];set yrange [-0.65:0.65]
gnuplot> set zrange [0:1.3]
gnuplot> t0=0;tf=3.5;dt=0.1
gnuplot> load "animate3D.gnu"
```

## 2.3 Trapped in a Box

In this section we will study the motion of a particle that is free, except when bouncing elastically on a wall or on certain obstacles. This motion is calculated by *approximate* algorithms that introduce systematic errors. These types of errors[15] are also encountered in the study of more complicated dynamics, but the simplicity of the problem will allow us to control them in a systematic and easy to understand way.

[15]In the previous sections, our calculations had a small systematic error due to the approximate nature of numerical floating point operations which approximate exact real number calculations. But the algorithms used were not introducing systematic errors like in the cases discussed in this section.

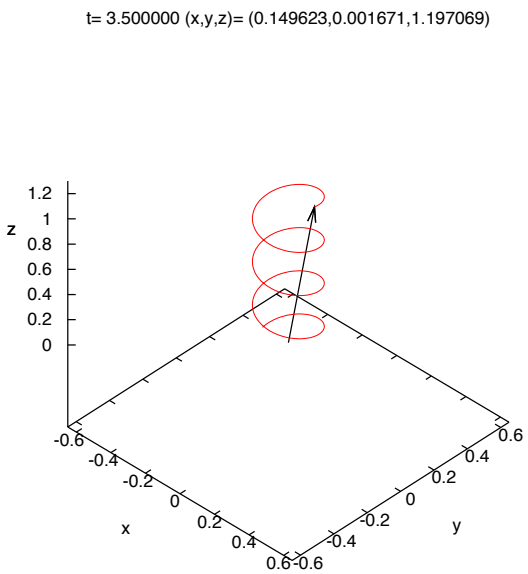



Figure 2.18: The trajectory $\vec{r}(t)$ calculated by the program in `ChargeInB.f90` for $\omega = 6.28$, $v_0 = 1.0$, $\theta = 20$ degrees as shown by the program `animate3D.gnu`. The current time and the coordinates of the particle are printed on the title of the plot.

### 2.3.1 The One Dimensional Box

The simplest example of such a motion is that of a particle in a "one dimensional box". The particle moves freely on the $x$ axis for $0 < x < L$, as can be seen in figure 2.19. When it reaches the boundaries $x = 0$ and $x = L$ it bounces and its velocity instantly reversed. Its potential energy is

$$V(x) = \begin{cases} 0 & 0 < x < L \\ +\infty & \text{elsewhere} \end{cases} , \tag{2.21}$$

which has the shape of an infinitely deep well. The force $F = -dV(x)/dx = 0$ within the box and $F = \pm\infty$ at the position of the walls.

Initially we have to know the position of the particle $x_0$ as well as its velocity $v_0$ (the sign of $v_0$ depends on the direction of the particle's motion) at time $t_0$. As long as the particle moves within the box, its motion is free and

$$\begin{aligned} x(t) &= x_0 + v_0(t - t_0) \\ v(t) &= v_0 \,. \end{aligned} \tag{2.22}$$

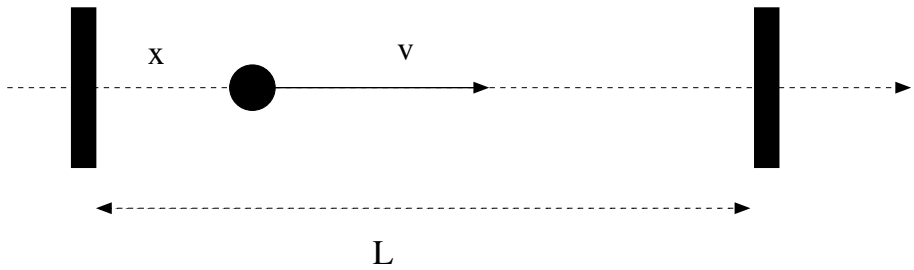



Figure 2.19: A particle in a one dimensional box with its walls located at $x = 0$ and $x = L$.

For a small enough change in time $\delta t$, so that there is no bouncing on the wall in the time interval $(t, t + \delta t)$, we have that

$$\begin{aligned} x(t+\delta t) &= x(t) + v(t)\delta t \\ v(t+\delta t) &= v(t)\,. \end{aligned} \tag{2.23}$$

Therefore we could use the above relations in our program and when the particle bounces off a wall we could simple reverse its velocity $v(t) \to -v(t)$. The devil is hiding in the word "when". Since the time interval $\delta t$ is finite in our program, there is no way to know the instant of the collision with accuracy better than $\sim \delta t$. However, our algorithm will change the direction of the velocity at time $t + \delta t$, when the particle will have already crossed the wall. This will introduce a systematic error, which is expected to decrease with decreasing $\delta t$. One way to implement the above idea is by constructing the loop

```
do while(t .le. tf)
 write(11,*)t,x,v
 x = x + v*dt
 t = t + dt
 if(x .lt. 0.0 .or. x .gt. L) v = -v
enddo
```

where the last line gives the testing condition for the wall collision and the subsequent change of the velocity.

The full program that realizes the proposed algorithm is listed below and can be found in the file `box1D_1.f90`. The user can set the size of the box `L`, the initial conditions `x0` and `v0` at time `t0`, the final time `tf` and the time step `dt`:

```
!==============================================================
!File box1D_1.f90
!Motion of a free particle in a box  0<x<L
!Use integration with time step dt: x = x + v*dt
!--------------------------------------------------------------
program box1D
 implicit none
!--------------------------------------------------------------
!Declaration of variables
 real ::  L,x0,v0,t0,tf,dt,t,x,v
!--------------------------------------------------------------
!Ask user for input:
 print *,'# Enter L:'
 read  *,L
 print *,'# L = ',L
 if( L .le. 0.0) stop 'L must be positive.'
 print *,'# Enter x0,v0:'
 read  *,x0,v0
 print *,'# x0= ',x0,' v0= ',v0
 if(x0 .lt. 0.0 .or. x0 .gt. L) stop 'illegal value of x0.'
 if(v0 .eq. 0.0                ) stop 'illegal value of v0 = 0.'
 print *,'# Enter t0,tf,dt:'
 read  *,t0,tf,dt
 print *,'# t0= ',t0,' tf= ',tf,' dt= ',dt
!--------------------------------------------------------------
!Initialize
 t = t0
 x = x0
 v = v0
 open(unit=11,file='box1D_1.dat')
!--------------------------------------------------------------
!Compute:
 do while(t .le. tf)
  write(11,*)t,x,v
  x = x + v*dt
  t = t + dt
  if(x .lt. 0.0 .or. x .gt. L) v = -v
 enddo
 close(11)
end program box1D
```

The computed data is recorded in the file `box1D_1.dat` in three columns. Compiling, running and plotting the trajectory using `gnuplot` can be done as follows:

```
> gfortran box1D_1.f90 -o box1
> ./box1
 # Enter L:
```

```
10
 # L =    10.
 # Enter x0,v0:
0 1.0
 # x0=    0. v0=    1.
 # Enter t0,tf,dt:
0 100 0.01
 # t0=    0. tf=    100. dt=    0.00999999978
> gnuplot
gnuplot> plot "box1D_1.dat" using  1:2 w l title "x(t)",\
                                    0 notitle,L notitle
gnuplot> plot [:][-1.2:1.2] "box1D_1.dat" \
                            using  1:3 w l title "v(t)"
```

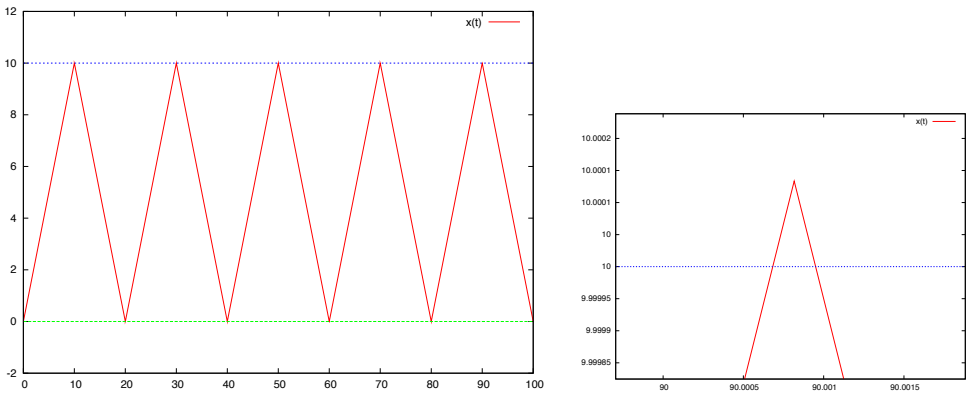



Figure 2.20: The trajectory $x(t)$ of a particle in a box with $L = 10$, $x_0 = 0.0$, $v_0 = 1.0$, $t_0 = 0$, $\delta t = 0.01$. The plot to the right magnifies a detail when $t \approx 90$ which exposes the systematic errors in determining the exact moment of the collision of the particle with the wall at $t_k = 90$ and the corresponding maximum value of $x(t)$, $x_m = L = 10.0$.

The trajectory $x(t)$ is shown in figure 2.20. The effects of the systematic errors can be easily seen by noting that the expected collisions occur every $T/2 = L/v = 10$ units of time. Therefore, on the plot to the right of figure 2.20, the reversal of the particle's motion should have occurred at $t = 90$, $x = L = 10$.

The reader should have already realized that the above mentioned error can be made to vanish by taking arbitrarily small $\delta t$. Therefore, we naively expect that as long as we have the necessary computer power to take $\delta t$ as small as possible and the corresponding time intervals as many as possible, we can achieve any precision that we want. Well, that is true only up to a point. The problem is that the next position is determined by the addition operation `x+v*dt` and the next moment in

time by `t+dt`. Floating point numbers of the `REAL` type have a maximum accuracy of approximately 7 significant decimal digits. Therefore, if the operands `x` and `v*dt` are real numbers differing by more than 7 orders of magnitude (`v*dt`$\lesssim 10^{-7}$ `x`), the effect of the addition `x+v*dt=x`, which is null! The reason is that the floating point unit of the processor has to convert both numbers `x` and `v*dt` into a representation having the same exponent and in doing so, the corresponding significant digits of the smaller number `v*dt` are lost. The result is less catastrophic when `v*dt`$\lesssim 10^{-a}$ `x` with $0 < a < 7$, but some degree of accuracy is also lost at each addition operation. And since we have accumulation of such errors over many intervals `t`$\rightarrow$`t+dt`, the error can become significant and destroy our calculation for large enough times. A similar error accumulates in the determination of the next instant of time `t+dt`, but we will discuss below how to make this contribution to the total error negligible. The above mentioned errors can become less detrimental by using floating point numbers of greater accuracy than the `REAL` type. For example `REAL(8)` numbers have approximately 16 significant decimal digits. But again, the precision is finite and the same type of errors are there only to be revealed by a more demanding and complicated calculation.

The remedy to such a problem can only be a change in the algorithm. This is not always possible, but in the case at hand this is easy to do. For example, consider the equation that gives the position of a particle in free motion

$$x(t) = x_0 + v_0(t - t_0)\,. \tag{2.24}$$

Let's use the above relation for the parts of the motion between two collisions. Then, all we have to do is to reverse the direction of the motion and reset the initial position and time to be the position and time of the collision. This can be done by using the loop:

```
t = t0
do while(t .le. tf)
 x = x0 + v0*(t-t0)
 write(11,*)t,x,v0
 if( x .lt. 0.0 .or. x .gt. L)then
  x0 =  x
  t0 =  t
  v0 = -v0
 endif
 t   =  t + dt
```

In the above algorithm, the error in the time of the collision is not van-

ishing but we don't have the "instability" problem of the dt$\rightarrow$ 0 limit[16]. Therefore we can isolate and study the effect of each type of error. The full program that implements the above algorithm is given below and can be found in the file box1D_2.f90:

```
!==========================================================
!File box1D_2.f90
!Motion of a free particle in a box  0<x<L
!Use constant velocity equation: x = x0 + v0*(t-t0)
!Reverse velocity and redefine x0,t0 on boundaries
!----------------------------------------------------------
program box1D
 implicit none
!----------------------------------------------------------
!Declaration of variables
 real ::  L,x0,v0,t0,tf,dt,t,x,v
!----------------------------------------------------------
!Ask user for input:
 print *,'# Enter L:'
 read  *,L
 print *,'# L = ',L
 if( L .le. 0.0) stop 'L must be positive.'
 print *,'# Enter x0,v0:'
 read  *,x0,v0
 print *,'# x0= ',x0,' v0= ',v0
 if(x0 .lt. 0.0 .or. x0 .gt. L) stop 'illegal value of x0.'
 if(v0 .eq. 0.0                ) stop 'illegal value of v0 = 0.'
 print *,'# Enter t0,tf,dt:'
 read  *,t0,tf,dt
 print *,'# t0= ',t0,' tf= ',tf,' dt= ',dt
!----------------------------------------------------------
!Initialize
 t = t0
 open(unit=11,file='box1D_2.dat')
!----------------------------------------------------------
!Compute:
 do while(t .le. tf)
  x = x0 + v0*(t-t0)
  write(11,*)t,x,v0
  if( x .lt. 0.0 .or. x .gt. L)then
   x0 =  x
   t0 =  t
   v0 = -v0
  endif
  t   =  t + dt
```

[16]We still have this problem in the t=t+dt operation. See discussion in the next section.

```
 enddo
 close(11)
end program box1D
```

Compiling and running the above program is done as before and the results are stored in the file `box1D_2.dat`. The algorithm can be improved in order to compute the exact solution. We leave that as an exercise for the reader[17].

### 2.3.2 Errors

In this section we will study the effect of the systematic errors that we encountered in the previous section in more detail. We considered two types of errors: First, the systematic error of determining the instant of the collision of the particle with the wall. This error is *reduced* by taking a smaller time step $\delta t$. Then, the systematic error that accumulates with each addition of two numbers with increasing difference in their orders of magnitude. This error is *increased* with decreasing $\delta t$. The competition of the two effects makes the optimal choice of $\delta t$ the result of a careful analysis. Such a situation is found in many interesting problems, therefore it is quite instructive to study it in more detail.

When the exact solution of the problem is not known, the systematic errors are controlled by studying the behavior of the solution as a function of $\delta t$. If the solutions are converging in a region of values of $\delta t$, one gains confidence that the true solution has been determined up to the accuracy of the convergence.

In the previous sections, we studied two different algorithms, programmed in the files `box1D_1.f90` and `box1D_2.f90`. We will refer to them as "method 1" and "method 2" respectively. We will study the convergence of the results as $\delta t \to 0$ by fixing all the parameters except $\delta t$ and then study the dependence of the results on $\delta t$. We will take $L = 10$, $v_0 = 1.0$, $x_0 = 0.0$, $t_0 = 0.0$, $t_f = 95.0$, so that the particle will collide with the wall every 10 units of time. We will measure the position of the particle $x(t \approx 95)$[18] as a function of $\delta t$ and study its convergence to a limit[19] as $\delta t \to 0$.

The analysis requires a lot of repetitive work: Compiling, setting the parameter values, running the program and calculating the value of $x(t \approx$

[17] See the file `box1D_3.dat`.
[18] Note the $\approx$!
[19] Of course we know the answer: $x(95) = 5$.

95) for many values of $\delta t$. We write the values of the parameters read by the program in a file `box1D_anal.in`:

```
10          L
0 1.0       x0 v0
0 95  0.05  t0 tf dt
```

Then we compile the program

```
> gfortran box1D_1.f90 -o box
```

and run it with the command:

```
> cat box1D_anal.in | ./box
```

By using the pipe `|`, we send the contents of `box1D_anal.in` to the `stdin` of the command `./box` by using the command `cat`. The result $x(t \approx 95)$ can be found in the last line of the file `box1D_1.dat`:

```
> tail -n 1  box1D_1.dat
  94.9511948  5.45000267 -1.
```

The third number in the above line is the value of the velocity. In a file `box1D_anal.dat` we write $\delta t$ and the first two numbers coming out from the command `tail`. Then we decrease the value $\delta t \to \delta t/2$ in the file `box1D_anal.in` and run again. We repeat for 12 more times until $\delta t$ reaches the value[20] 0.000012. We do the same[21] using method 2 and we place the results for $x(t \approx 95)$ in two new columns in the file `box1D_anal.dat`. The result is

```
# ------------------------------------------------
#  dt      t1_95     x1(95)    x2(95)
# ------------------------------------------------
0.050000 94.95119 5.450003 5.550126
0.025000 94.97849 5.275011 5.174837
0.012500 94.99519 5.124993 5.099736
0.006250 94.99850 4.987460 5.063134
0.003125 94.99734 5.021894 5.035365
```

[20]Try the command `sed 's/0.05/0.025/' box1D_anal.in | ./box` 0.000012 by changing `0.025` with the desired value of $\delta t$.

[21]See the shell script `box1D_anal.csh` as a suggestion on how to automate this boring process.

```
0.001563 94.99923 5.034538 5.017764
0.000781 94.99939 4.919035 5.011735
0.000391 94.99979 4.695203 5.005493
0.000195 95.00000 5.434725 5.001935
0.000098 94.99991 5.528124 5.000745
0.000049 94.99998 3.358000 5.000330
0.000024 94.99998 2.724212 5.000232
0.000012 94.99999 9.240705 5.000158
```

Convergence is studied in figure 2.21. The 1st method maximizes its accuracy for $\delta t \approx 0.01$, whereas for $\delta t < 0.0001$ the error becomes $> 10\%$ and the method becomes useless. The 2nd method has much better behavior that the 1st one.

We observe that as $\delta t$ decreases, the final value of $t$ approaches the expected $t_f = 95$. Why don't we obtain $t = 95$, especially when $t/\delta t$ is an integer? How many steps does it really take to reach $t \approx 95$, when the expected number of those is $\approx 95/\delta t$? Each time you take a measurement, issue the command

```
> wc -l box1D_1.dat
```

which measures the number of lines in the file `box1D_1.dat` and compare this number with the expected one. The result is interesting:

```
# ------------------------------
#    dt     N          N0
# ------------------------------
0.050000 1900      1900
0.025000 3800      3800
0.012500 7601      7600
0.006250 15203     15200
0.003125 30394     30400
0.001563 60760     60780
0.000781 121751    121638
0.000391 243753    242966
0.000195 485144    487179
0.000098 962662    969387
0.000049 1972589   1938775
0.000024 4067548   3958333
0.000012 7540956   7916666
```

where the second column has the number of steps computed by the program and the third one has the expected number of steps. We observe that the accuracy decreases with decreasing $\delta t$ and in the end the difference is about 5%! Notice that the last line should have given

$t_f = 0.000012 \times 7540956 \approx 90.5$, an error comparable to the period of the particle's motion.

We conclude that one important source of accumulation of systematic errors is the calculation of time. This type of errors become more significant with decreasing $\delta t$. We can improve the accuracy of the calculation significantly if we use the multiplication `t=t0+i*dt` instead of the addition `t=t+dt`, where `i` is a step counter:

```
!t = t  + dt      ! Not accurate,    avoid
 t = t0 + i*dt    ! Better accuracy, prefer
```

The main loop in the program `box1D_1.f90` becomes:

```
 t = t0
 x = x0
 v = v0
 i = 0
 do while(t .le. tf)
  write(11,*)t,x,v
  i = i  + 1
  x = x  + v*dt
  t = t0 + i*dt
  if(x .lt. 0.0 .or. x .gt. L) v = -v
 enddo
```

The full program can be found in the file `box1D_4.f90` of the accompanying software. We call this "method 3". We perform the same change in the file `box1D_2.f90`, which we store in the file `box1D_5.f90`. We call this "method 4". We repeat the same analysis using methods 3 and 4 and we find that the problem of calculating time accurately practically vanishes. The result of the analysis can be found on the right plot of figure 2.21. Methods 2 and 4 have no significant difference in their results, whereas methods 1 and 3 do have a dramatic difference, with method 3 decreasing the error more than tenfold. The problem of the increase of systematic errors with decreasing $\delta t$ does not vanish completely due to the operation `x=x+v*dt`. This type of error is harder to deal with and one has to invent more elaborate algorithms in order to reduce it significantly. This will be discussed further in chapter 4.

### 2.3.3 The Two Dimensional Box

A particle is confined to move on the plane in the area $0 < x < L_x$ and $0 < y < L_y$. When it reaches the boundaries of this two dimensional

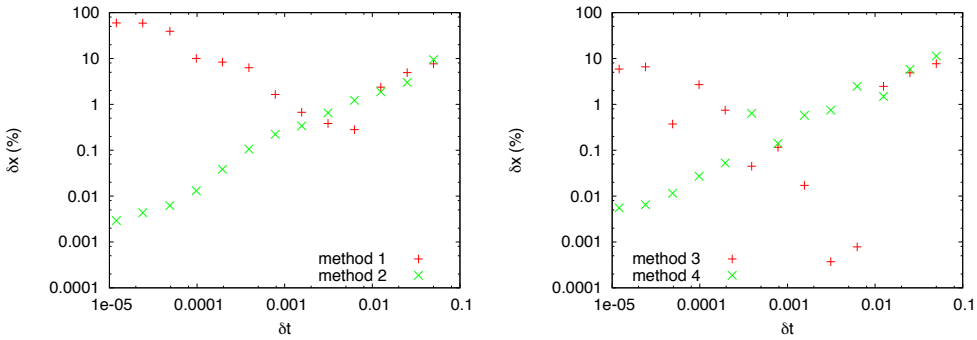



Figure 2.21: The error $\delta x = 2|x_i(95) - x(95)|/|x_i(95) + x(95)| \times 100$ where $x_i(95)$ is the value calculated by method $i = 1, 2, 3, 4$ and $x(95)$ the exact value according to the text.

box, it bounces elastically off its walls. The particle is found in an infinite depth orthogonal potential well. The particle starts moving at time $t_0$ from $(x_0, y_0)$ and our program will calculate its trajectory until time $t_f$ with time step $\delta t$. Such a trajectory can be seen in figure 2.23.

If the particle's position and velocity are known at time $t$, then at time $t + \delta t$ they will be given by the relations

$$\begin{aligned} x(t+\delta t) &= x(t) + v_x(t)\delta t \\ y(t+\delta t) &= y(t) + v_y(t)\delta t \\ v_x(t+\delta t) &= v_x(t) \\ v_y(t+\delta t) &= v_y(t)\,. \end{aligned} \tag{2.25}$$

The collision of the particle off the walls is modeled by reflection of the normal component of the velocity when the respective coordinate of the particle crosses the wall. This is a source of the systematic errors that we discussed in the previous section. The central loop of the program is:

```
i = i  + 1
t = t0 + i *dt
x = x  + vx*dt
y = y  + vy*dt
if(x .lt. 0.0 .or. x .gt. Lx) vx = -vx
if(y .lt. 0.0 .or. y .gt. Ly) vy = -vy
```

The full program can be found in the file `box2D_1.f90`. Notice that we introduced two counters `nx` and `ny` of the particle's collisions with the walls:

```
!=============================================================
!File box2D_1.f90
!Motion of a free particle in a box  0<x<Lx 0<y<Ly
!Use integration with time step dt: x = x + vx*dt y=y+vy*dt
!-------------------------------------------------------------
program box2D
 implicit none
!-------------------------------------------------------------
!Declaration of variables
 real(8) ::  Lx,Ly,x0,y0,v0x,v0y,t0,tf,dt,t,x,y,vx,vy
 integer ::  i,nx,ny
!-------------------------------------------------------------
!Ask user for input:
 print *,'# Enter Lx,Ly:'
 read  *,Lx,Ly
 print *,'# Lx = ',Lx,' Ly= ',Ly
 if( Lx .le. 0.0) stop 'Lx must be positive.'
 if( Ly .le. 0.0) stop 'Ly must be positive.'
 print *,'# Enter x0,y0,v0x,v0y:'
 read  *,x0,y0,v0x,v0y
 print *,'# x0= ',x0,' y0= ',y0,' v0x= ',v0x,' v0y= ',v0y
 if(x0 .lt. 0.0 .or. x0 .gt. Lx) stop 'illegal value x0'
 if(y0 .lt. 0.0 .or. y0 .gt. Ly) stop 'illegal value y0'
 if(v0x**2+v0y**2.eq. 0.0      ) stop 'illegal value v0=0'
 print *,'# Enter t0,tf,dt:'
 read  *,t0,tf,dt
 print *,'# t0= ',t0,' tf= ',tf,' dt= ',dt
!-------------------------------------------------------------
!Initialize
 i  = 0
 nx = 0  ;  ny = 0
 t  = t0
 x  = x0 ;  y  = y0
 vx = v0x;  vy = v0y
 open(unit=11,file='box2D_1.dat')
!-------------------------------------------------------------
!Compute:
 do while(t .le. tf)
  write(11,*)t,x,y,vx,vy
  i = i  + 1
  t = t0 + i *dt
  x = x  + vx*dt
  y = y  + vy*dt
  if(x .lt. 0.0 .or. x .gt. Lx) then
   vx = -vx
   nx =  nx + 1
  endif
  if(y .lt. 0.0 .or. y .gt. Ly) then
```

```
    vy = -vy
    ny =  ny + 1
   endif
  enddo
  close(11)
  print *,'# Number of collisions:'
  print *,'# nx= ',nx,' ny= ',ny
end program box2D
```

A typical session for the study of a particle's trajectory could be:

```
> gfortran box2D_1.f90 -o box
> ./box
 # Enter Lx,Ly:
10.0 5.0
 # Lx =   10. Ly=   5.
 # Enter x0,y0,v0x,v0y:
5.0 0.0 1.27 1.33
 # x0=   5. y0=   0. v0x=   1.27 v0y=   1.33
 # Enter t0,tf,dt:
0 50 0.01
 # t0=   0. tf=   50. dt=   0.01
 # Number of collisions:
 # nx=  6 ny=  13
> gnuplot
gnuplot> plot   "box2D_1.dat" using 1:2 w l title "x (t)"
gnuplot> replot "box2D_1.dat" using 1:3 w l title "y (t)"
gnuplot> plot   "box2D_1.dat" using 1:4 w l title "vx(t)"
gnuplot> replot "box2D_1.dat" using 1:5 w l title "vy(t)"
gnuplot> plot   "box2D_1.dat" using 2:3 w l title "x-y"
```

Note the last line of output from the program: The particle bounces off the vertical walls 6 times (`nx=6`) and from the horizontal ones 13 (`ny=13`). The `gnuplot` commands construct the diagrams displayed in figures 2.22 and 2.23.

In order to animate the particle's trajectory, we can copy the file `box2D_animate.gnu` of the accompanying software to the current directory and give the `gnuplot` commands:

```
gnuplot> file = "box2D_1.dat"
gnuplot> Lx = 10 ; Ly = 5
gnuplot> t0 = 0  ; tf = 50; dt = 1
gnuplot> load "box2D_animate.gnu"
gnuplot> t0 = 0  ; dt = 0.5; load "box2D_animate.gnu"
```

The last line repeats the same animation at half speed. You can also

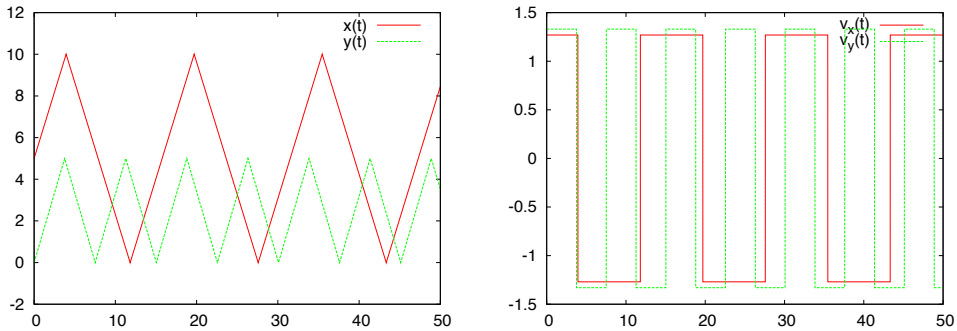



Figure 2.22: The results for the trajectory of a particle in a two dimensional box given by the program `box2D_1.f90`. The parameters are $L_x = 10$, $L_y = 5$, $x_0 = 5$, $y_0 = 0$, $v_{0x} = 1.27$, $v_{0y} = 1.33$, $t_0 = 0$, $t_f = 50$, $\delta t = 0.01$.

use the file `animate2D.gnu` discussed in section 2.1.1. We add new commands in the file `box2D_animate.gnu` so that the plot limits are calculated automatically and the box is drawn on the plot. The arrow drawn is not the position vector with respect to the origin of the coordinate axes, but the one connecting the initial with the current position of the particle.

The next step should be to test the accuracy of your results. This can be done by generalizing the discussion of the previous section and is left as an exercise for the reader.

## 2.4 Applications

In this section we will study simple examples of motion in a box with different types of obstacles. We will start with a game of ... mini golf. The player shoots a (point) "ball" which moves in an orthogonal box of linear dimensions $L_x$ and $L_y$ and which is open on the $x = 0$ side. In the box there is a circular "hole" with center at $(x_c, y_c)$ and radius $R$. If the "ball" falls in the "hole", the player wins. If the ball leaves out of the box through its open side, the player loses. In order to check if the ball is in the hole when it is at position $(x, y)$, all we have to do is to check whether $(x - x_c)^2 + (y - y_c)^2 \leq R^2$.

Initially we place the ball at the position $(0, L_y/2)$ at time $t_0 = 0$. The player hits the ball which leaves with initial velocity of magnitude $v_0$ at an angle $\theta$ degrees with the $x$ axis. The program is found in the file `MiniGolf.f90` and is listed below:

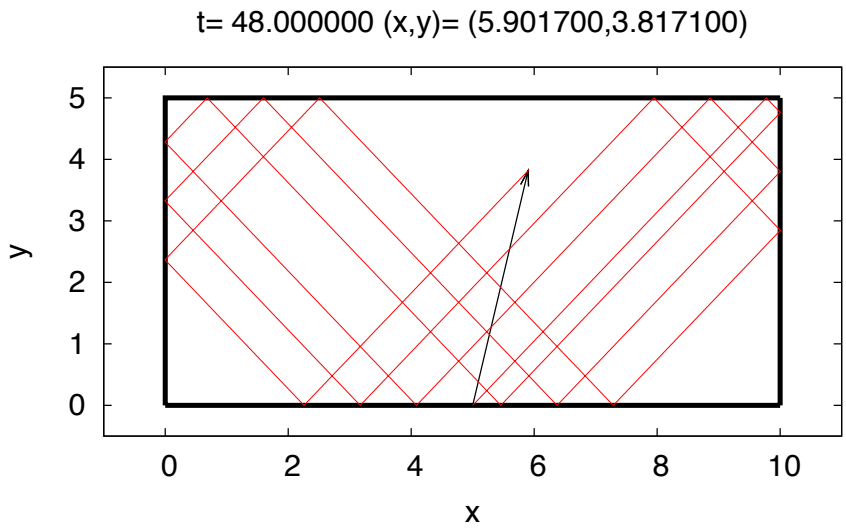



Figure 2.23: The trajectory of the particle of figure 2.22 until $t = 48$. The origin of the arrow is at the initial position of the particle and its end is at its current position. The bold lines mark the boundaries of the box.

```
!==========================================================
!File MiniGolf.f
!Motion of a free particle in a box  0<x<Lx 0<y<Ly
!The box is open at x=0 and has a hole at (xc,yc) of radius R
!Ball is shot at (0,Ly/2) with speed v0, angle theta (degrees)
!Use integration with time step dt: x = x + vx*dt y=y+vy*dt
!Ball stops in hole (success) or at x=0 (failure)
!----------------------------------------------------------
program MiniGolf
 implicit none
!----------------------------------------------------------
!Declaration of variables
 real(8)              :: Lx,Ly,x0,y0,v0x,v0y,t0,tf,dt,t,x,y,vx,vy
 real(8)              :: v0,theta,xc,yc,R,R2
 real(8), parameter   :: PI=3.14159265358979324D0
 integer              :: i,nx,ny
 character(7)         :: result
!----------------------------------------------------------
!Ask user for input:
 print *,'# Enter Lx,Ly:'
 read  *,Lx,Ly
 print *,'# Lx = ',Lx,' Ly= ',Ly
```

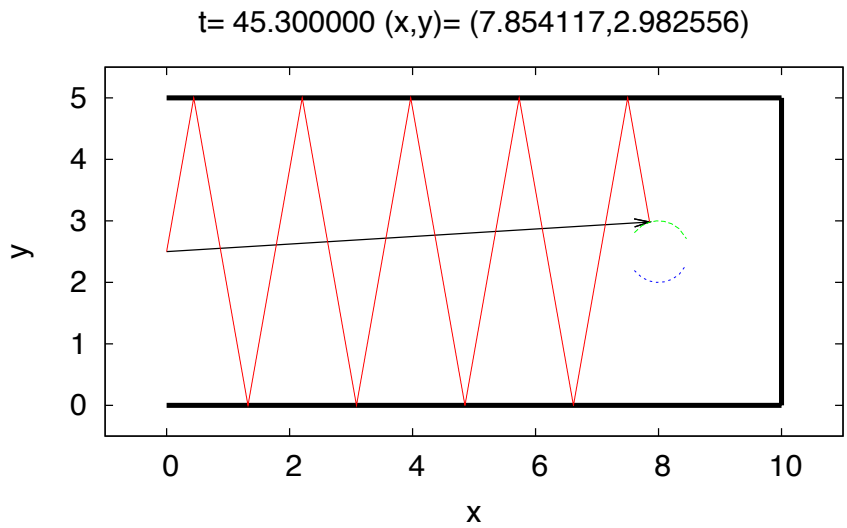



Figure 2.24: The trajectory of the particle calculated by the program `MiniGolf.f90` using the parameters chosen in the text. The moment of ... success is shown. At time $t = 45.3$ the particle enters the hole's region which has its center at $(8, 2.5)$ and its radius is $0.5$.

```
 if( Lx .le. 0.0) stop 'Lx must be positive.'
 if( Ly .le. 0.0) stop 'Ly must be positive.'
 print *,'# Enter hole position and radius: (xc,yc), R:'
 read  *,xc,yc,R
 print *,'# (xc,yc)= ( ',xc,' , ',yc,' ) R= ',R
 print *,'# Enter v0, theta(degrees):'
 read  *,v0,theta
 print *,'# v0= ',v0,' theta= ',theta,' degrees'
 if(v0        .le. 0.0D0 ) stop 'illegal value of v0.'
 if(ABS(theta).ge. 90.0D0) stop 'illegal value of theta.'
 print *,'# Enter dt:'
 read  *,dt
 print *,'# dt= ',dt
!-----------------------------------------------------------------
!Initialize
 t0 = 0.0D0
 x0 = 0.00001D0 ! small but non-zero
 y0 = Ly/2.0
 R2 = R*R
 theta = (PI/180.0D0)*theta
 v0x = v0*cos(theta)
```

```
 v0y = v0*sin(theta)
 print *,'# x0= ',x0,' y0= ',y0,' v0x= ',v0x,' v0y= ',v0y
 i  = 0
 nx = 0  ;  ny = 0
 t  = t0
 x  = x0 ;  y  = y0
 vx = v0x;  vy = v0y
 open(unit=11,file='MiniGolf.dat')
!-----------------------------------------------------------------
!Compute:
 do while( .TRUE. ) !forever!
  write(11,*)t,x,y,vx,vy
  i = i  + 1
  t = t0 + i*dt
  x = x  + vx*dt
  y = y  + vy*dt
  if(x .gt. Lx) then
   vx = -vx
   nx =  nx + 1
  endif
  if(y .lt. 0.0 .or. y .gt. Ly) then
   vy = -vy
   ny = ny  + 1
  endif
  if(x .le. 0.0D0)then
   result = 'Failure'
   exit !exit do loop
  endif
  if( ((x-xc)*(x-xc)+(y-yc)*(y-yc)) .le. R2)then
   result = 'Success'
   exit !exit do loop
  endif
 enddo
 close(11)
 print *,'# Number of collisions:'
 print *,'# Result= ',result,' nx= ',nx,' ny= ',ny
end program MiniGolf
```

In order to run it, we can use the commands:

```
> gfortran MiniGolf.f90 -o mg
> ./mg
 # Enter Lx,Ly:
10 5
 # Lx =   10. Ly=   5.
 # Enter hole position and radius: (xc,yc), R:
8 2.5 0.5
 # (xc,yc)= (   8. ,   2.5 ) R=   0.5
```

```
 # Enter v0, theta(degrees):
1 80
 # v0=   1. theta=   80. degrees
 # Enter dt:
0.01
 # dt=   0.01
 # x0=   1.E-05 y0=   2.5 v0x=   0.173648178 v0y=   0.984807753
 # Number of collisions:
 # Result= Success nx=  0 ny=  9
```

You should construct the plots of the position and the velocity of the particle. You can also use the animation program found in the file `MiniGolf_animate.gnu` for fun. Copy it from the accompanying software to the current directory and give the `gnuplot` commands:

```
gnuplot> file = "MiniGolf.dat"
gnuplot> Lx = 10;Ly = 5
gnuplot> xc = 8; yc = 2.5 ; R = 0.5
gnuplot> t0 = 0; dt = 0.1
gnuplot> load "MiniGolf_animate.gnu"
```

The results are shown in figure 2.24.

The next example with be three dimensional. We will study the motion of a particle confined within a cylinder of radius $R$ and height $L$. The collisions of the particle with the cylinder are elastic. We take the axis of the cylinder to be the $z$ axis and the two bases of the cylinder to be located at $z = 0$ and $z = L$. This is shown in figure 2.26.

The collisions of the particle with the bases of the cylinder are easy to program: we follow the same steps as in the case of the simple box. For the collision with the cylinder's side, we consider the projection of the motion on the $x - y$ plane. The projection of the particle moves within a circle of radius $R$ and center at the intersection of the $z$ axis with the plane. This is shown in figure 2.25. At the collision, the $r$ component of the velocity is reflected $v_r \to -v_r$, whereas $v_\theta$ remains the same. The velocity of the particle before the collision is

$$\begin{aligned} \vec{v} &= v_x\hat{x} + v_y\hat{y} \\ &= v_r\hat{r} + v_\theta\hat{\theta} \end{aligned} \tag{2.26}$$

and after the collision is

$$\begin{aligned} \vec{v}' &= v'_x\hat{x} + v'_y\hat{y} \\ &= -v_r\hat{r} + v_\theta\hat{\theta} \end{aligned} \tag{2.27}$$

From the relations

$$\begin{aligned} \hat{r} &= \cos\theta\hat{x} + \sin\theta\hat{y} \\ \hat{\theta} &= -\sin\theta\hat{x} + \cos\theta\hat{y}\,, \end{aligned} \tag{2.28}$$

and $v_r = \vec{v}\cdot\hat{r}$, $v_\theta = \vec{v}\cdot\hat{\theta}$, we have that

$$\begin{aligned} v_r &= v_x\cos\theta + v_y\sin\theta \\ v_\theta &= -v_x\sin\theta + v_y\cos\theta\,. \end{aligned} \tag{2.29}$$

The inverse relations are

$$\begin{aligned} v_x &= v_r\cos\theta - v_\theta\sin\theta \\ v_y &= v_r\sin\theta + v_\theta\cos\theta\,. \end{aligned} \tag{2.30}$$

With the transformation $v_r \to -v_r$, the new velocity in Cartesian coordinates will be

$$\begin{aligned} v'_x &= -v_r\cos\theta - v_\theta\sin\theta \\ v'_y &= -v_r\sin\theta + v_\theta\cos\theta\,. \end{aligned} \tag{2.31}$$

The transformation $v_x \to v'_x$, $v_y \to v'_y$ will be performed in the subroutine `reflectVonCircle(vx,vy,x,y,xc,yc,R)`. Upon entry to the subroutine, we provide the initial velocity `(vx,vy)`, the collision point `(x,y)`, the center of the circle `(xc,yc)` and the radius of the circle[22] `R`. Upon exit from the subroutine, `(vx,vy)` have been replaced with the new values[23] $(v'_x, v'_y)$.

The program can be found in the file `Cylinder3D.f90` and is listed below:

```
!==========================================================
!File Cylinder3D.f90
!Motion of a free particle in a cylinder with axis the z-axis,
!radius R and 0<z<L
!Use integration with time step dt: x = x + vx*dt
!                                   y = y + vy*dt
!                                   z = z + vz*dt
!Use subroutine reflectVonCircle for collisions at r=R
!----------------------------------------------------------
program Cylinder3D
```

[22]Of course one expects $R^2 = (x-x_c)^2 + (y-y_c)^2$, but because of systematic errors, we require $R$ to be given.

[23]Note that upon exit, the particle is also placed exactly on the circle.

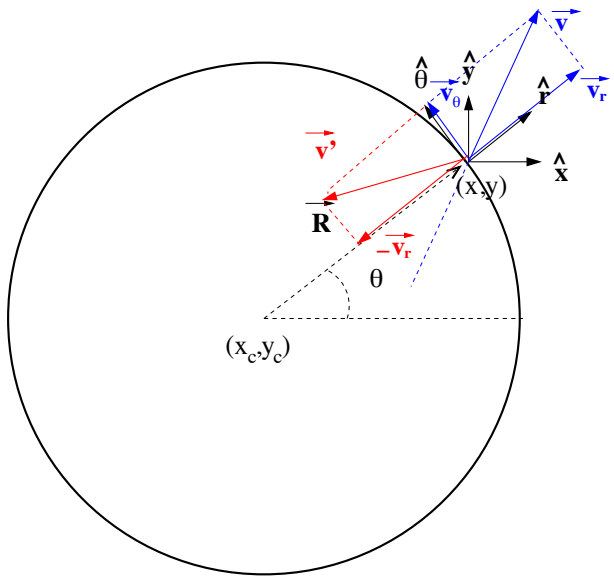



Figure 2.25: The elastic collision of the particle moving within the circle of radius $R = |\vec{R}|$ and center $\vec{r}_c = x_c\hat{x} + y_c\hat{y}$ at the point $\vec{r} = x\hat{x} + y\hat{y}$. We have that $\vec{R} = (x - x_c)\hat{x} + (y - y_c)\hat{y}$. The initial velocity is $\vec{v} = v_r\hat{r} + v_\theta\hat{\theta}$ where $\hat{r} \equiv \vec{R}/R$. After reflecting $v_r \to -v_r$ the new velocity of the particle is $\vec{v}' = -v_r\hat{r} + v_\theta\hat{\theta}$.

```
 implicit none
!-----------------------------------------------------------------
!Declaration of variables
 real(8)   ::  x0,y0,z0,v0x,v0y,v0z,t0,tf,dt,t,x,y,z,vx,vy,vz
 real(8)   ::  L,R,R2,vxy,rxy,r2xy,xc,yc
 integer   ::  i,nr,nz
!-----------------------------------------------------------------
!Ask user for input:
 print *,'# Enter R, L:'
 read  *,R,L
 print *,'# R= ',R,' L= ',L
 if( R .le. 0.0) stop 'R must be positive.'
 if( L .le. 0.0) stop 'L must be positive.'
 print *,'# Enter x0,y0,z0,v0x,v0y,v0z:'
 read  *,x0,y0,z0,v0x,v0y,v0z
 rxy = DSQRT(x0*x0+y0*y0)
 print *,'# x0 = ',x0 ,' y0 = ',y0 ,' z0=  ',z0, ' rxy= ',rxy
 print *,'# v0x= ',v0x,' v0y= ',v0y,' v0z= ',v0z
 if(rxy .gt. R                 )stop 'illegal value of rxy > R'
 if(z0  .lt. 0.0D0             )stop 'illegal value of z0  < 0'
 if(z0  .gt. L                 )stop 'illegal value of z0  > L'
 if(v0x**2+v0y**2+v0z**2.eq.0.0)stop 'illegal value of v0 = 0.'
 print *,'# Enter t0,tf,dt:'
```

```
 read  *,t0,tf,dt
 print *,'# t0= ',t0,' tf= ',tf,' dt= ',dt
!-----------------------------------------------------------
!Initialize
 i  = 0
 nr = 0  ;  nz = 0
 t  = t0
 x  = x0 ;  y  = y0 ;  z  = z0
 vx = v0x; vy  = v0y;  vz = v0z
 R2 = R*R
 xc = 0.0D0 !center of circle which is the projection of the
 yc = 0.0D0 !cylinder on the xy plane
 open(unit=11,file='Cylinder3D.dat')
!-----------------------------------------------------------
!Compute:
 do while(t .le. tf)
  write(11,100)t,x,y,z,vx,vy,vz
  i = i  + 1
  t = t0 + i *dt
  x = x  + vx*dt
  y = y  + vy*dt
  z = z  + vz*dt
  if(z .lt. 0.0 .or. z .gt. L) then
   vz = -vz            ! reflection on cylinder caps
   nz =  nz + 1
  endif
  r2xy = x*x+y*y
  if( r2xy .gt. R2)then
   call reflectVonCircle(vx,vy,x,y,xc,yc,R)
   nr  = nr + 1
  endif
 enddo
 close(11)
 print *,'# Number of collisions:'
 print *,'# nr= ',nr,' nz= ',nz

100 FORMAT(100G28.16)
end program Cylinder3D
!-----------------------------------------------------------
!===========================================================
!-----------------------------------------------------------
subroutine reflectVonCircle(vx,vy,x,y,xc,yc,R)
 implicit none
 real(8)  :: vx,vy,x,y,xc,yc,R
 real(8)  :: theta,cth,sth,vr,vth

 theta = atan2(y-yc,x-xc)
 cth   = cos(theta)
 sth   = sin(theta)
```

```
 vr     =  vx*cth + vy *sth
 vth    = -vx*sth + vy *cth

 vx     = -vr*cth - vth*sth !reflect vr -> -vr
 vy     = -vr*sth + vth*cth

 x      =  xc     + R*cth   !put x,y on the circle
 y      =  yc     + R*sth
end subroutine reflectVonCircle
```

Note that the function `atan2` is used for computing the angle `theta`. This function, when called with two arguments `atan2(y,x)`, returns the angle $\theta = \tan^{-1}(y/x)$ in radians. The correct quadrant of the circle where $(x, y)$ lies is chosen. The angle that we want to compute is given by `atan2(y-yc,x-xc)`. Then we apply equations (2.29) and (2.31) and in the last two lines we enforce the particle to be at the point $(x_c + R\cos\theta, y_c + R\sin\theta)$, exactly on the circle.

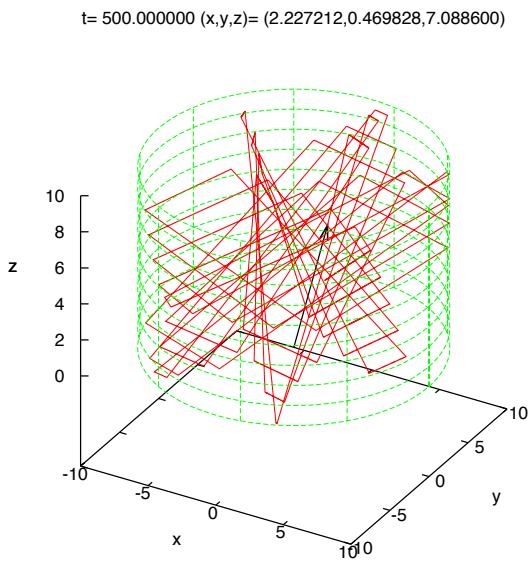



Figure 2.26: The trajectory of a particle moving inside a cylinder with $R = 10$, $L = 10$, computed by the program `Cylinder3D.f90`. We have chosen $\vec{r}_0 = 1.0\hat{x} + 2.2\hat{y} + 3.1\hat{z}$, $\vec{v}_0 = 0.93\hat{x} - 0.89\hat{y} + 0.74\hat{z}$, $t_0 = 0$, $t_f = 500.0$, $\delta t = 0.01$.

A typical session is shown below:

```
> gfortran Cylinder3D.f90 -o cl
> ./cl
 # Enter R, L:
10.0 10.0
 # R=   10. L=   10.
 # Enter x0,y0,z0,v0x,v0y,v0z:
1.0 2.2 3.1   0.93 -0.89 0.74
 # x0 =   1. y0 =   2.2 z0=    3.1 rxy=   2.41660919
 # v0x=   0.93 v0y=  -0.89 v0z=   0.74
 # Enter t0,tf,dt:
0.0 500.0 0.01
 # t0=   0. tf=   500. dt=   0.01
 # Number of collisions:
 # nr=  33 nz=  37
```

In order to plot the position and the velocity as a function of time, we use the following `gnuplot` commands:

```
gnuplot> file="Cylinder3D.dat"
gnuplot> plot file using 1:2 with lines title "  x(t)",\
             file using 1:3 with lines title "  y(t)",\
             file using 1:4 with lines title "  z(t)"
gnuplot> plot file using 1:5 with lines title "v_x(t)",\
             file using 1:6 with lines title "v_y(t)",\
             file using 1:7 with lines title "v_z(t)"
```

We can also compute the distance of the particle from the cylinder's axis $r(t) = \sqrt{x(t)^2 + y(t)^2}$ as a function of time using the command:

```
gnuplot> plot file using 1:(sqrt($2**2+$3**2)) w l t "r(t)"
```

In order to plot the trajectory, together with the cylinder, we give the commands:

```
gnuplot> L = 10 ; R = 10
gnuplot> set urange [0:2.0*pi]
gnuplot> set vrange [0:L]
gnuplot> set parametric
gnuplot> splot file using 2:3:4 with lines notitle,\
                      R*cos(u),R*sin(u),v notitle
```

The command `set parametric` is necessary if one wants to make a parametric plot of a surface $\vec{r}(u,v) = x(u,v)\,\hat{x} + y(u,v)\,\hat{y} + z(u,v)\,\hat{z}$. The cylinder (without the bases) is given by the parametric equations $\vec{r}(u,v) = R\cos u\,\hat{x} + R\sin u\,\hat{y} + v\,\hat{z}$ with $u \in [0, 2\pi)$, $v \in [0, L]$.

We can also animate the trajectory with the help of the `gnuplot` script file `Cylinder3D_animate.gnu`. Copy the file from the accompanying software to the current directory and give the `gnuplot` commands:

```
gnuplot> R=10;L=10;t0=0;tf=500;dt=10
gnuplot> load "Cylinder3D_animate.gnu"
```

The result is shown in figure 2.26.

The last example will be that of a simple model of a spacetime wormhole. This is a simple spacetime geometry which, in the framework of the theory of general relativity, describes the connection of two distant areas in space which are asymptotically flat. This means, that far enough from the wormhole's mouths, space is almost flat - free of gravity. Such a geometry is depicted in figure 2.27. The distance traveled by someone through the mouths could be much smaller than the distance traveled outside the wormhole and, at least theoretically, traversable wormholes could be used for interstellar/intergalactic traveling and/or communications between otherwise distant areas in the universe. Of course we should note that such macroscopic and stable wormholes are not known to be possible to exist in the framework of general relativity. One needs an exotic type of matter with negative energy density which has never been observed. Such exotic geometries may realize microscopically as quantum fluctuations of spacetime and make the small scale structure of the geometry[24] a "spacetime foam".

We will study a very simple model of the above geometry on the plane with a particle moving freely in it[25]. We take the two dimensional plane and cut two equal disks of radius $R$ with centers at distance $d$ like in figure 2.28. We identify the points on the two circles such that the point 1 of the left circle is the same as the point 1 on the right circle, the point 2 on the left with the point 2 on the right etc. The two circles are given by the parametric equations $x(\theta) = d/2 + R\cos\theta$, $y(\theta) = R\sin\theta$, $-\pi < \theta \leq \pi$ for the right circle and $x(\theta) = -d/2 - R\cos\theta$, $y(\theta) = R\sin\theta$, $-\pi < \theta \leq \pi$ for the left. Points on the two circles with the same $\theta$ are identified. A particle entering the wormhole from the left circle with velocity $v$ is immediately exiting from the right with velocity $v'$ as shown in figure 2.28.

[24]See K.S. Thorne "Black Holes and Time Wraps: Einstein's Outrageous Legacy", W.W. Norton, New York for a popular review of these concepts.

[25]This idea can be found as an exercise in the excellent introductory general relativity textbook J. B. Hartle, "Gravity: An Introduction to Einstein's General Relativity", Addison Wesley 2003, Ch. 7, Ex. 25.

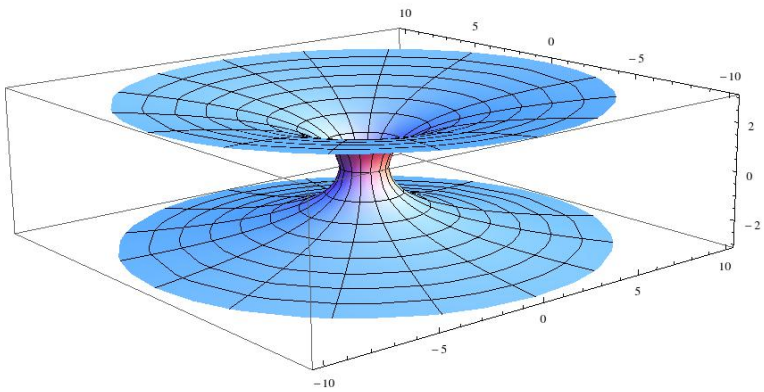



Figure 2.27: A typical geometry of space near a wormhole. Two asymptotically flat regions of space are connected through a "neck" which can be arranged to be of small length compared to the distance of the wormhole mouths when traveled from the outside space.

Then we will do the following:

1. Write a program that computes the trajectory of a particle moving in the geometry of figure 2.28. We set the limits of motion to be $-L/2 \leq x \leq L/2$ and $-L/2 \leq y \leq L/2$. We will use periodic boundary conditions in order to define what happens when the particle attempts to move outside these limits. This means that we identify the $x = -L/2$ line with the $x = +L/2$ line as well as the $y = -L/2$ line with the $y = +L/2$ line. The user enters the parameters $R$, $d$ and $L$ as well as the initial conditions $(x_0, y_0)$, $(v_0, \phi)$ where $\vec{v}_0 = v_0(\cos\phi\hat{x} + \sin\phi\hat{y})$. The user will also provide the time parameters $t_f$ and $dt$ for motion in the time interval $t \in [t_0 = 0, t_f]$ with step $dt$.

2. Plot the particle's trajectory with $(x_0, y_0) = (0, -1)$, $(v_0, \phi) = (1, 10^o)$ με $t_f = 40$, $dt = 0.05$ in the geometry with $L = 20, d = 5, R = 1$.

3. Find a closed trajectory which does not cross the boundaries $|x| = L/2$, $|y| = L/2$ and determine whether it is stable under small perturbations of the initial conditions.

4. Find other closed trajectories that go through the mouths of the

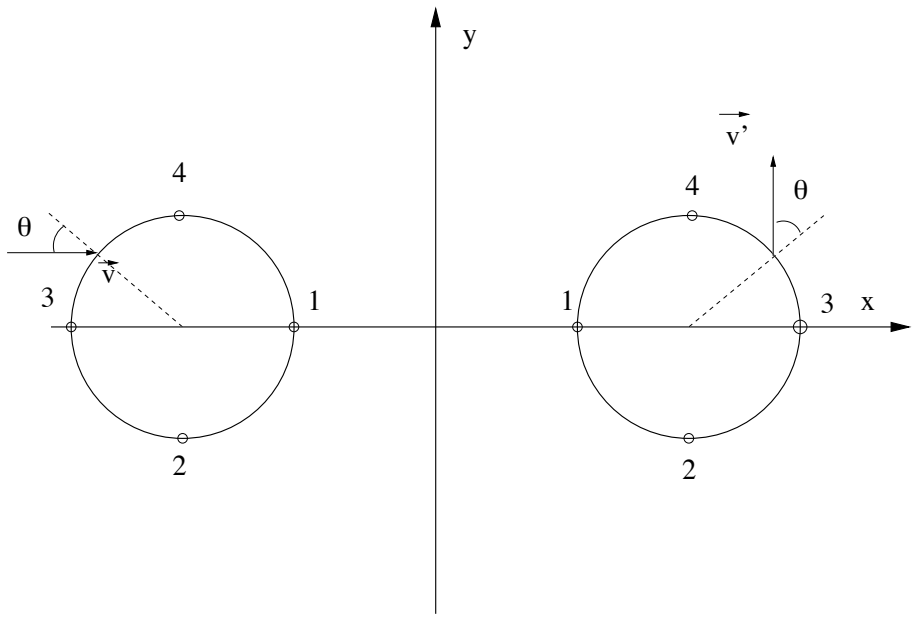



Figure 2.28: A simple model of the spacetime geometry of figure 2.27. The particle moves on the whole plane except withing the two disks that have been removed. The neck of the wormhole is modeled by the two circles $x(\theta) = \pm d/2 \pm R\cos\theta$, $y(\theta) = R\sin\theta$, $-\pi < \theta \leq \pi$ and has zero length since their points have been identified. There is a given direction in this identification, so that points with the same $\theta$ are the same (you can imagine how this happens by folding the plane across the $y$ axis and then glue the two circles together). The entrance of the particle through one mouth and exit through the other is done as shown for the velocity vector $\vec{v} \to \vec{v}'$.

wormhole and study their stability under small perturbations of the initial conditions.

5. Add to the program the option to calculate the distance traveled by the particle. If the particle starts from $(-x_0, 0)$ and moves in the $+x$ direction to the $(x_0, 0)$, $x_0 > R + d/2$ position, draw the trajectory and calculate the distance traveled on paper. Then confirm your calculation from the numerical result coming from your program.

6. Change the boundary conditions, so that the particle bounces off elastically at $|x| = L/2$, $|y| = L/2$ and replot all the trajectories mentioned above.

Define the right circle $c_1$ by the parametric equations

$$x(\theta) = \frac{d}{2} + R\cos\theta\,, \qquad y(\theta) = R\sin\theta\,, \qquad -\pi < \theta \leq \pi\,, \tag{2.32}$$

and the left circle $c_2$ by the parametric equations

$$x(\theta) = -\frac{d}{2} - R\cos\theta\,, \qquad y(\theta) = R\sin\theta\,, \qquad -\pi < \theta \leq \pi\,. \tag{2.33}$$

The particle's position changes at time $dt$ by

$$\begin{aligned} t_i &= idt \\ x_i &= x_{i-1} + v_x dt \\ y_i &= y_{i-1} + v_y dt \end{aligned} \tag{2.34}$$

for $i = 1, 2, \ldots$ for given $(x_0, y_0)$, $t_0 = 0$ and as long as $t_i \leq t_f$. If the point $(x_i, y_i)$ is outside the boundaries $|x| = L/2$, $|y| = L/2$, we redefine $x_i \to x_i \pm L$, $y_i \to y_i \pm L$ in each case respectively. Points defined by the same value of $\theta$ are identified, i.e. they represent the *same* points of space. If the point $(x_i, y_i)$ crosses either one of the circles $c_1$ or $c_2$, then we take the particle out from the other circle.

Crossing the circle $c_1$ is determined by the relation

$$\left(x_i - \frac{d}{2}\right)^2 + y_i^2 \leq R^2\,. \tag{2.35}$$

The angle $\theta$ is calculated from the equation

$$\theta = \tan^{-1}\left(\frac{y_i}{x_i - \frac{d}{2}}\right)\,, \tag{2.36}$$

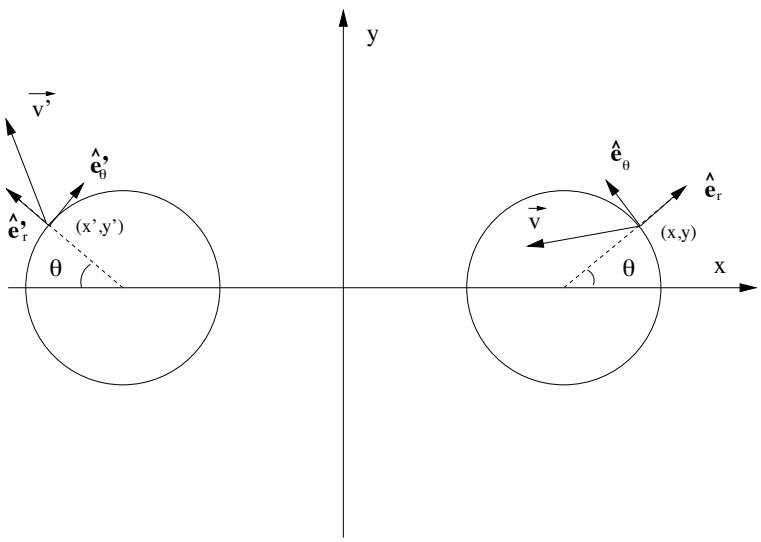



Figure 2.29: The particle crossing the wormhole through the right circle $c_1$ with velocity $\vec{v}$. It emerges from $c_2$ with velocity $\vec{v}'$. The unit vectors $(\hat{e}_r, \hat{e}_\theta)$, $(\hat{e}'_r, \hat{e}'_\theta)$ are computed from the parametric equations of the two circles $c_1$ and $c_2$.

and the point $(x_i, y_i)$ is mapped to the point $(x'_i, y'_i)$ where

$$x'_i = -\frac{d}{2} - R\cos\theta\,, \qquad y'_i = y_i\,, \tag{2.37}$$

as can be seen in figure 2.29. For mapping $\vec{v} \to \vec{v}'$, we first calculate the vectors

$$\left.\begin{array}{rrcr} \hat{e}_r = & \cos\theta\,\hat{x} & + & \sin\theta\,\hat{y} \\ \hat{e}_\theta = & -\sin\theta\,\hat{x} & + & \cos\theta\,\hat{y} \end{array}\right\} \to \left\{\begin{array}{rrcr} \hat{e}'_r = & -\cos\theta\,\hat{x} & + & \sin\theta\,\hat{y} \\ \hat{e}'_\theta = & \sin\theta\,\hat{x} & + & \cos\theta\,\hat{y} \end{array}\right. , \tag{2.38}$$

so that the velocity

$$\vec{v} = v_r\,\hat{e}_r + v_\theta\,\hat{e}_\theta \quad \to \quad \vec{v}' = -v_r\,\hat{e}'_r + v_\theta\,\hat{e}'_\theta\,, \tag{2.39}$$

where the radial components are $v_r = \vec{v}\cdot\hat{e}_r$ and $v_\theta = \vec{v}\cdot\hat{e}_\theta$. Therefore, the relations that give the "emerging" velocity $\vec{v}'$ are:

$$\begin{array}{rrcr} v_r = & v_x\cos\theta & + & v_y\sin\theta \\ v_\theta = & -v_x\sin\theta & + & v_y\cos\theta \\ v'_x = & v_r\cos\theta & + & v_\theta\sin\theta \\ v'_y = & -v_r\sin\theta & + & v_\theta\cos\theta \end{array}\,. \tag{2.40}$$

Similarly we calculate the case of entering from $c_2$ and emerging from $c_1$. The condition now is:

$$\left(x_i + \frac{d}{2}\right)^2 + y_i^2 \le R^2\,. \tag{2.41}$$

The angle $\theta$ is given by

$$\theta = \pi - \tan^{-1}\left(\frac{y_i}{x_i + \frac{d}{2}}\right) , \tag{2.42}$$

and the point $(x_i, y_i)$ is mapped to the point $(x_i', y_i')$ where

$$x_i' = \frac{d}{2} + R\cos\theta\,, \qquad y_i' = y_i\,. \tag{2.43}$$

For mapping $\vec{v} \to \vec{v}'$, we calculate the vectors

$$\left.\begin{array}{rrcr} \hat{e}_r = & -\cos\theta\,\hat{x} & + & \sin\theta\,\hat{y} \\ \hat{e}_\theta = & \sin\theta\,\hat{x} & + & \cos\theta\,\hat{y} \end{array}\right\} \to \left\{\begin{array}{rrcr} \hat{e}_r' = & \cos\theta\,\hat{x} & + & \sin\theta\,\hat{y} \\ \hat{e}_\theta' = & -\sin\theta\,\hat{x} & + & \cos\theta\,\hat{y} \end{array}\right. , \tag{2.44}$$

so that the velocity

$$\vec{v} = v_r\,\hat{e}_r + v_\theta\,\hat{e}_\theta \quad \to \quad \vec{v}' = -v_r\,\hat{e}_r' + v_\theta\,\hat{e}_\theta'\,. \tag{2.45}$$

The emerging velocity $\vec{v}'$ is:

$$\begin{array}{rrcr} v_r = & -v_x\cos\theta & + & v_y\sin\theta \\ v_\theta = & v_x\sin\theta & + & v_y\cos\theta \\ v_x' = & -v_r\cos\theta & - & v_\theta\sin\theta \\ v_y' = & -v_r\sin\theta & + & v_\theta\cos\theta \end{array} . \tag{2.46}$$

Systematic errors are now coming from crossing the two mouths of the wormhole. There are no systematic errors from crossing the boundaries $|x| = L/2$, $|y| = L/2$ (why?). Try to think of ways to control those errors and study them.

The closed trajectories that we are looking for come from the initial conditions

$$(x_0, y_0, v_0, \phi) = (0, 0, 1, 0) \tag{2.47}$$

and they connect points 1 of figure 2.28. They are unstable, as can be seen by taking $\phi \to \phi + \epsilon$.

The closed trajectories that cross the wormhole and "wind" through space can come from the initial conditions

$$\begin{array}{rcl} (x_0, y_0, v_0, \phi) & = & (-9, 0, 1, 0) \\ (x_0, y_0, v_0, \phi) & = & (2.5, -3, 1, 90^o) \end{array}$$

and cross the points $3 \to 3$ and $2 \to 2 \to 4 \to 4$ respectively. They are also unstable, as can be easily verified by using the program that you will write. The full program is listed below:

```
!==========================================================
program WormHole2D
 implicit none
!----------------------------------------------------------
!Declaration of variables
 real(8), parameter :: PI=3.14159265358979324D0
 real(8) ::  Lx,Ly,L,R,d
 real(8) ::  x0,y0,v0,theta
 real(8) ::  t0,tf,dt
 real(8) ::  t,x,y,vx,vy
 real(8) ::  xc1,yc1,xc2,yc2,r1,r2
 integer ::  i
!----------------------------------------------------------
!Ask user for input:
 print *,'# Enter L,d,R:'
 read  *,L,d,R
 print *,'# L= ',L,' d= ',d,' R= ',R
 if( L .le. d+2.0D0*R) stop 'L <= d+2*R'
 if( d .le.   2.0D0*R) stop 'd <=   2*R'
 print *,'# Enter (x0,y0), v0, theta(degrees):'
 read  *,x0,y0,v0,theta
 print *,'# x0= ',x0,' y0   = ',y0
 print *,'# v0= ',v0,' theta= ',theta,' degrees'
 if(v0 .le.   0.0D0  ) stop 'illegal value of v0.'
 print *,'# Enter tf, dt:'
 read  *,tf,dt
 print *,'# tf= ',tf,' dt= ',dt
!----------------------------------------------------------
!Initialize
 theta = (PI/180.0D0)*theta
 i     =  0
 t     =  0.0D0
 x     =  x0           ; y     =  y0
 vx    =  v0*cos(theta); vy    =  v0*sin(theta)
 print *,'# x0= ',x,' y0= ',y,' v0x= ',vx,' v0y= ',vy
!Wormhole's centers:
 xc1   =  0.5D0*d; yc1   =  0.0D0
 xc2   = -0.5D0*d; yc2   =  0.0D0
!Box limits coordinates:
 Lx    =  0.5D0*L; Ly    =  0.5D0*L
!Test if already inside cut region:
 r1    = sqrt((x-xc1)**2+(y-yc1)**2)
 r2    = sqrt((x-xc2)**2+(y-yc2)**2)
 if( r1    .le. R ) stop 'r1 <= R'
 if( r2    .le. R ) stop 'r2 <= R'
!Test if outside box limits:
 if(ABS(x) .ge. Lx) stop '|x| >= Lx'
 if(ABS(y) .ge. Ly) stop '|y| >= Ly'
```

```
 open(unit=11,file='Wormhole.dat')
!-----------------------------------------------------------------
!Compute:
 do while( t .lt. tf )
  write(11,*)t,x,y,vx,vy
  i  = i+1
  t  = i*dt
  x  = x + vx*dt; y  = y + vy*dt
! Toroidal boundary conditions:
  if( x .gt.  Lx) x  = x - L
  if( x .lt. -Lx) x  = x + L
  if( y .gt.  Ly) y  = y - L
  if( y .lt. -Ly) y  = y + L
! Test if inside the cut disks
  r1 = sqrt((x-xc1)**2+(y-yc1)**2)
  r2 = sqrt((x-xc2)**2+(y-yc2)**2)
  if( r1 .lt. R)then
! Notice: we pass r1 as radius of circle, not R
   call crossC1(x,y,vx,vy,dt,r1,d)
  else if( r2 .lt. R)then
   call crossC2(x,y,vx,vy,dt,r2,d)
  endif
! small chance here that still in C1 or C2, but OK since
! another dt-advance given at the beginning of do-loop
 enddo !do while( t .lt. tf )
end program WormHole2D
!=========================================================
subroutine crossC1(x,y,vx,vy,dt,R,d)
 implicit none
 real(8) ::  x,y,vx,vy,dt,R,d
 real(8) ::  vr,v0 !v0 -> vtheta
 real(8) ::  theta,xc,yc
 print *,'# Inside C1: (x,y,vx,vy,R)= ',x,y,vx,vy,R
 xc    =  0.5D0*d !center of C1
 yc    =  0.0D0
 theta =  atan2(y-yc,x-xc)
 x     = -xc - R*cos(theta) !new x-value, y invariant
!Velocity transformation:
 vr    =  vx*cos(theta)+vy*sin(theta)
 v0    = -vx*sin(theta)+vy*cos(theta)
 vx    =  vr*cos(theta)+v0*sin(theta)
 vy    = -vr*sin(theta)+v0*cos(theta)
!advance x,y, hopefully outside C2:
 x     = x + vx*dt
 y     = y + vy*dt
 print *,'# Exit   C2: (x,y,vx,vy  )= ',x,y,vx,vy
end subroutine crossC1
!=========================================================
subroutine crossC2(x,y,vx,vy,dt,R,d)
```

```
 implicit none
 real(8), parameter :: PI=3.14159265358979324D0
 real(8) ::  x,y,vx,vy,dt,R,d
 real(8) ::  vr,v0 !v0 -> vtheta
 real(8) ::  theta,xc,yc

 print *,'# Inside C2: (x,y,vx,vy,R)= ',x,y,vx,vy,R
 xc    = -0.5D0*d !center of C2
 yc    =  0.0D0
 theta =  PI-atan2(y-yc,x-xc)
 x     = -xc + R*cos(theta) !new x-value, y invariant
!Velocity transformation:
 vr    = -vx*cos(theta)+vy*sin(theta)
 v0    =  vx*sin(theta)+vy*cos(theta)
 vx    = -vr*cos(theta)-v0*sin(theta)
 vy    = -vr*sin(theta)+v0*cos(theta)
!advance x,y, hopefully outside C1:
 x     = x + vx*dt
 y     = y + vy*dt
 print *,'# Exit   C1: (x,y,vx,vy  )= ',x,y,vx,vy
end subroutine crossC2
```

It is easy to compile and run the program. See also the files `Wormhole.csh` and `Wormhole_animate.gnu` of the accompanying software and run the `gnuplot` commands:

```
gnuplot> file  = "Wormhole.dat"
gnuplot> R=1;d=5;L=20;
gnuplot> ! ./Wormhole.csh
gnuplot> t0=0;dt=0.2;load "Wormhole_animate.gnu"
```

You are now ready to answer the rest of the questions that we asked in our list.

## 2.5 Problems

2.1 Change the program `Circle.f90` so that it prints the number of full circles traversed by the particle.

2.2 Add all the necessary tests on the parameters entered by the user in the program `Circle.f90`, so that the program is certain to run without problems. Do the same for the rest of the programs given in the same section.

2.3 A particle moves with constant angular velocity $\omega$ on a circle that has the origin of the coordinate system at its center. At time $t_0 = 0$, the particle is at $(x_0, y_0)$. Write the program `CircularMotion.f90` that will calculate the particle's trajectory. The user should enter the parameters $\omega, x_0, y_0, t_0, t_f, \delta t$. The program should print the results like the program `Circle.f90` does.

2.4 Change the program `SimplePendulum.f90` so that the user could enter a non zero initial velocity.

2.5 Study the $k \to 0$ limit in the projectile motion given by equations (2.10). Expand $\mathrm{e}^{-kt} = 1 - kt + \frac{1}{2!}(kt)^2 + \ldots$ and keep the non vanishing terms as $k \to 0$. Then keep the next order leading terms which have a smaller power of $k$. Program these relations in a file `ProjectileSmallAirResistance.f90`. Consider the initial conditions $\vec{v}_0 = \hat{x} + \hat{y}$ and calculate the range of the trajectory numerically by using the two programs `ProjectileSmallAirResistance.f90`, `ProjectileAirResistance.f90`. Determine the range of values of $k$ for which the two results agree within 5% accuracy.

2.6 Write a program for a projectile which moves through a fluid with fluid resistance proportional to the square of the velocity. Compare the range of the trajectory with the one calculated by the program `ProjectileAirResistance.f90` for the parameters shown in figure 2.10.

2.7 Change the program `Lissajous.f90` so that the user can enter a different amplitude and initial phase in each direction. Study the case where the amplitudes are the same and the phase difference in the two directions are $\pi/4, \pi/2, \pi, -\pi$. Repeat by taking the amplitude in the $y$ direction to be twice as much the amplitude in the $x$ direction.

2.8 Change the program `ProjectileAirResistance.f90`, so that it can calculate also the $k = 0$ case.

2.9 Change the program `ProjectileAirResistance.f90` so that it can calculate the trajectory of the particle in three dimensional space. Plot the position coordinates and the velocity components as a function of time. Plot the three dimensional trajectory using `splot` in `gnuplot` and animate the trajectory using the `gnuplot` script `animate3D.gnu`.

2.10 Change the program `ChargeInB.f90` so that it can calculate the number of full revolutions that the projected particle's position on the $x - y$ plane makes during its motion.

2.11 Change the program `box1D_1.f90` so that it prints the number of the particle's collisions on the left wall, on the right wall and the total number of collisions to the `stdout`.

2.12 Do the same for the program `box1D_2.f90`. Fill the table on page 115 the number of calculated collisions and comment on the results.

2.13 Run the program `box1D_1.f90` and choose `L= 10, v0=1`. Decrease the step `dt` up to the point that the particle stops to move. For which value of `dt` this happens? Increase `v0=10,100`. Until which value of `dt` the particle moves now? Why?

2.14 Change the `REAL` declarations to `REAL(8)` in the program `box1D_1.f90`. Add explicit exponents `D0` to all constants (e.g. `0.0`→`0.0D0`). Compare your results to those obtained in section 2.3.2. Repeat problem 2.13. What do you observe?

2.15 Change the program `box1D_1.f90` so that you can study non elastic collisions $v' = -ev$, $0 < e \leq 1$ with the walls.

2.16 Change the program `box2D_1.f90` so that you can study inelastic collisions with the walls, such that $v'_x = -ev_x$, $v'_y = -ev_y$, $0 < e \leq 1$.

2.17 Use the method of calculating time in the programs `box1D_4.f90` and `box1D_5.f90` in order to produce the results in figure 2.21.

2.18 Particle falls freely moving in the vertical direction. It starts with zero velocity at height $h$. Upon reaching the ground, it bounces inelastically such that $v'_y = -ev_y$ with $0 < e \leq 1$ a parameter. Write the necessary program in order to study numerically the particle's motion and study the cases $e = 0.1, 0.5, 0.9, 1.0$.

2.19 Generalize the program of the previous problem so that you can study the case $\vec{v}_0 = v_{0x}\,\hat{x}$. Animate the calculated trajectories.

2.20 Study the motion of a particle moving inside the box of figure 2.30. Count the number of collisions of the particle with the walls before it leaves the box.

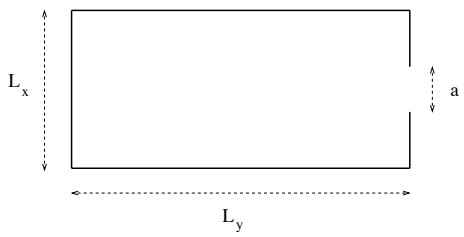



Figure 2.30: Problem 2.20.

2.21 Study the motion of the point particle on the "billiard table" of figure 2.31. Count the number of collisions with the walls before the particle enters into a hole. The program should print from which hole the particle left the table.

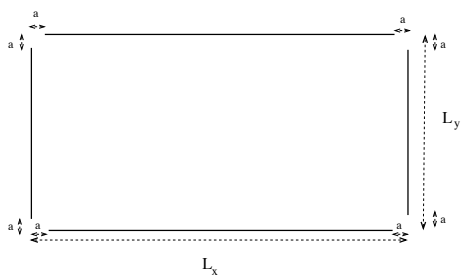



Figure 2.31: Problem 2.21.

2.22 Write a program in order to study the motion of a particle in the box of figure 2.32. At the center of the box there is a disk on which the particle bounces off elastically (Hint: use the routine `reflectVonCircle` of the program `Cylinder3D.f90`).

2.23 In the box of the previous problem, put four disks on which the particle bounces of elastically like in figure 2.33.

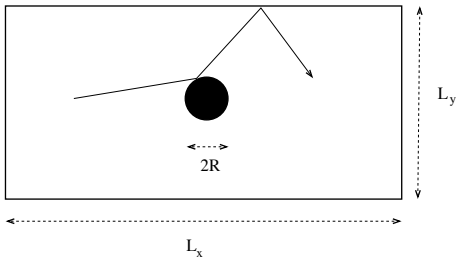



Figure 2.32: Problem 2.22.

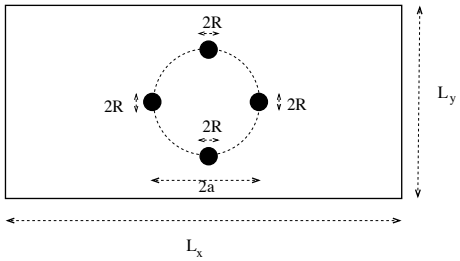



Figure 2.33: Problem 2.23.

2.24 Consider the arrangement of figure 2.34. Each time the particle bounces elastically off a circle, the circle disappears. The game is over successfully if all the circles vanish. Each time the particle bounces off on the wall to the left, you lose a point. Try to find trajectories that minimize the number of lost points.

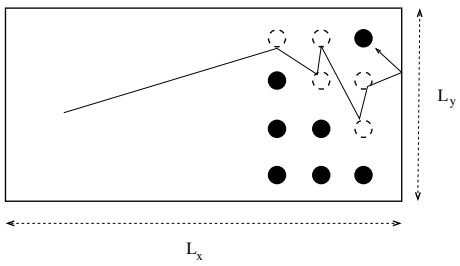



Figure 2.34: Problem 2.24.

# Chapter 3

# Logistic Map

Nonlinear differential equations model interesting dynamical systems in physics, biology and other branches of science. In this chapter we perform a numerical study of the discrete logistic map as a "simple mathematical model with complex dynamical properties" [18] similar to the ones encountered in more complicated and interesting dynamical systems. For certain values of the parameter of the map, one finds chaotic behavior giving us an opportunity to touch on this very interesting topic with important consequences in physical phenomena. Chaotic evolution restricts out ability for *useful* predictions in an otherwise fully deterministic dynamical system: measurements using slightly different initial conditions result in a distribution which is indistinguishable from the distribution coming from sampling a random process. This scientific field is huge and active and we refer the reader to the bibliography for a more complete introduction [18, 19, 20, 21, 22, 23, 24, 35].

## 3.1 Introduction

The most celebrated application of the logistic map comes from the study of population growth in biology. One considers populations which reproduce at fixed time intervals and whose generations do not overlap.

The simplest (and most naive) model is the one that makes the reasonable assumption that the rate of population growth $dP(t)/dt$ of a population $P(t)$ is proportional to the current population:

$$\frac{dP(t)}{dt} = kP(t)\,. \tag{3.1}$$

The general solution of the above equation is $P(t) = P(0)\mathrm{e}^{kt}$ showing an *exponential* population growth for $k > 0$ an decline for $k < 0$. It

is obvious that this model is reasonable as long as the population is small enough so that the interaction with its environment (adequate food, diseases, predators etc) can be neglected. The simplest model that takes into account some of the factors of the interaction with the environment (e.g. starvation) is obtained by the introduction of a simple non linear term in the equation so that

$$\frac{dP(t)}{dt} = kP(t)(1 - bP(t)) \,. \tag{3.2}$$

The parameter $k$ gives the maximum growth rate of the population and $b$ controls the ability of the species to maintain a certain population level. The equation (3.2) can be discretized in time by assuming that each generation reproduces every $\delta t$ and that the n-th generation has population $P_n = P(t_n)$ where $t_n = t_0 + (n-1)\delta t$. Then $P(t_{n+1}) \approx P(t_n) + \delta t P'(t_n)$ and equation (3.1) becomes

$$P_{n+1} = rP_n \,, \tag{3.3}$$

where $r = 1 + k\delta t$. The solutions of the above equation are well approximated by $P_n \sim P_0 \mathrm{e}^{kt_n} \propto \mathrm{e}^{(r-1)n}$ so that we have population growth when $r > 1$ and decline when $r < 1$. Equation (3.2) can be discretized as follows:

$$P_{n+1} = P_n(r - bP_n) \,. \tag{3.4}$$

Defining $x_n = (b/r)P_n$ we obtain the logistic map

$$x_{n+1} = rx_n(1 - x_n) \,. \tag{3.5}$$

We define the functions

$$f(x) = rx(1 - x), \qquad F(x, r) = rx(1 - x) \tag{3.6}$$

(their only difference is that, in the first one, $r$ is considered as a given parameter), so that

$$x_{n+1} = f(x_n) = f^{(2)}(x_{n-1}) = \ldots = f^{(n)}(x_1) = f^{(n+1)}(x_0) \,, \tag{3.7}$$

where we use the notation $f^{(1)}(x) = f(x)$, $f^{(2)}(x) = f(f(x))$, $f^{(3)}(x) = f(f(f(x)))$, ... for function composition. In what follows, the derivative of $f$ will be useful:

$$f'(x) = \frac{\partial F(x, r)}{\partial x} = r(1 - 2x) \,. \tag{3.8}$$

Since we interpret $x_n$ to be the fraction of the population with respect to its maximum value, we should have $0 \leq x_n \leq 1$ for each[1] $n$. The function $f(x)$ has one global maximum for $x = 1/2$ which is equal to $f(1/2) = r/4$. Therefore, if $r > 4$, then $f(1/2) > 1$, which for an appropriate choice of $x_0$ will lead to $x_{n+1} = f(x_n) > 1$ for some value of $n$. Therefore, the interval of values of $r$ which is of interest for our model is

$$0 < r \leq 4\,. \tag{3.9}$$

The logistic map (3.5) may be viewed as a finite difference equation and it is a one step inductive relation. Given an initial value $x_0$, a sequence of values $\{x_0,\, x_1,\, \ldots,\, x_n,\, \ldots\}$ is produced. This will be referred[2] to as the *trajectory* of $x_0$. In the following sections we will study the properties of these trajectories as a function of the parameter $r$.

The solutions of the logistic map are not known except in special cases. For $r = 2$ we have

$$x_n = \frac{1}{2}\left(1 - (1 - x_0)^{2n}\right)\,, \tag{3.10}$$

and for[3] $r = 4$

$$x_n = \sin^2(2^n \pi\theta)\,, \qquad \theta = \frac{1}{\pi}\sin^{-1}\sqrt{x_0}\,. \tag{3.11}$$

For $r = 2$, $\lim_{n\to\infty} x_n = 1/2$ whereas for $r = 4$ we have periodic trajectories resulting in rational $\theta$ and non periodic resulting in irrational $\theta$. For other values of $r$ we have to resort to a numerical computation of the trajectories of the logistic map.

## 3.2 Fixed Points and $2^n$ Cycles

It is obvious that if the point $x^*$ is a solution of the equation $x = f(x)$, then $x_n = x^* \Rightarrow x_{n+k} = x^*$ for every $k \geq 0$. For the function $f(x) = rx(1-x)$ we have two solutions

$$x_1^* = 0 \qquad \text{και} \qquad x_2^* = 1 - 1/r\,. \tag{3.12}$$

[1]Note that if $x_n > 1$ then $x_{n+1} < 0$, so that if we want $x_n \geq 0$ for each $n$, then we should have $x_n \leq 1$ for each $n$.

[2]In the bibliography, the term "splinter of $x_0$" is frequently used.

[3]E. Schröder, "Über iterierte Funktionen", Math. Ann. **3** (1870) 296; E. Lorenz, "The problem of deducing the climate from the governing equations", Tellus **16** (1964) 1

We will see that for appropriate values of $r$, these solutions are *attractors* of most of the trajectories. This means that for a range of values for the initial point $0 \leq x_0 \leq 1$, the sequence $\{x_n\}$ approaches asymptotically one of these points as $n \to \infty$. Obviously the (measure zero) sets of initial values $\{x_0\} = \{x_1^*\}$ and $\{x_0\} = \{x_2^*\}$ result in trajectories attracted by $x_1^*$ and $x_2^*$ respectively. In order to determine which one of the two values is preferred, we need to study the *stability* of the fixed points $x_1^*$ and $x_2^*$. For this, assume that for some value of $n$, $x_n$ is infinitesimally close to the fixed point $x^*$ so that

$$\begin{aligned} x_n &= x^* + \epsilon_n \\ x_{n+1} &= x^* + \epsilon_{n+1} \,. \end{aligned} \tag{3.13}$$

Since

$$x_{n+1} = f(x_n) = f(x^* + \epsilon_n) \approx f(x^*) + \epsilon_n f'(x^*) = x^* + \epsilon_n f'(x^*) \,, \tag{3.14}$$

where we used the Taylor expansion of the analytic function $f(x^* + \epsilon_n)$ about $x^*$ and the relation $x^* = f(x^*)$, we have that $\epsilon_{n+1} = \epsilon_n f'(x^*)$. Then we obtain

$$\left| \frac{\epsilon_{n+1}}{\epsilon_n} \right| = |f'(x^*)| \,. \tag{3.15}$$

Therefore, if $|f'(x^*)| < 1$ we obtain $\lim_{n\to\infty} \epsilon_n = 0$ and the fixed point $x^*$ is *stable*: the sequence $\{x_{n+k}\}$ approaches $x^*$ asymptotically. If $|f'(x^*)| > 1$ then the sequence $\{x_{n+k}\}$ deviates away from $x^*$ and the fixed point is *unstable*. The limiting case $|f'(x^*)| = 1$ should be studied separately and it indicates a change in the stability properties of the fixed point. In the following discussion, these points will be shown to be bifurcation points.

For the function $f(x) = rx(1-x)$ with $f'(x) = r(1-2x)$ we have that $f'(0) = r$ and $f'(1-1/r) = 2-r$. Therefore, if $r < 1$ the point $x_1^* = 0$ is an attractor, whereas the point $x_2^* = 1 - 1/r < 0$ is irrelevant. When $r > 1$, the point $x_1^* = 0$ results in $|f'(x_1^*)| = r > 1$, therefore $x_1^*$ is unstable. Any initial value $x_0$ near $x_1^*$ deviates from it. Since for $1 < r < 3$ we have that $0 \leq |f'(x_2)| = |2-r| < 1$, the point $x_2^*$ is an attractor. Any initial value $x_0 \in (0,1)$ approaches $x_2^* = 1 - 1/r$. When $r = r_c^{(1)} = 1$ we have the limiting case $x_1^* = x_2^* = 0$ and we say that at the critical value $r_c^{(1)} = 1$ the fixed point $x_1^*$ *bifurcates* to the two fixed points $x_1^*$ and $x_2^*$.

As $r$ increases, the fixed points continue to bifurcate. Indeed, when $r = r_c^{(2)} = 3$ we have that $f'(x_2^*) = 2 - r = -1$ and for $r > r_c^{(2)}$ the point $x_2^*$ becomes unstable. Consider the solution of the equation $x = f^{(2)}(x)$. If $0 < x^* < 1$ is one of its solutions and for some $n$ we have that $x_n = x^*$,

then $x_{n+2} = x_{n+4} = \ldots = x_{n+2k} = \ldots = x^*$ and $x_{n+1} = x_{n+3} = \ldots = x_{n+2k+1} = \ldots = f(x^*)$ (therefore $f(x^*)$ is also a solution). If $0 < x_3^* < x_4^* < 1$ are two such *different* solutions with $x_3^* = f(x_4^*)$, $x_4^* = f(x_3^*)$, then the trajectory is periodic with period 2. The points $x_3^*$, $x_4^*$ are such that they are *real* solutions of the equation

$$f^{(2)}(x) = r^2 x(1-x)(1-rx(1-x)) = x\,, \tag{3.16}$$

and at the same time they are *not* the solutions $x_1^* = 0$ $x_2^* = 1 - 1/r$ of the equation[4] $x = f^{(2)}(x)$, the polynomial above can be written in the form (see [19] for more details)

$$x\left(x - \left(1 - \frac{1}{r}\right)\right)(Ax^2 + Bx + C) = 0\,. \tag{3.17}$$

By expanding the polynomials (3.16), (3.17) and comparing their coefficients we conclude that $A = -r^3$, $B = r^2(r+1)$ and $C = -r(r+1)$. The roots of the trinomial in (3.17) are determined by the discriminant $\Delta = r^2(r+1)(r-3)$. For the values of $r$ of interest ($1 < r \leq 4$), the discriminant becomes positive when $r > r_c^{(2)} = 3$ and we have two different solutions

$$x_\alpha^* = ((r+1) \mp \sqrt{r^2 - 2r - 3})/(2r) \qquad \alpha = 3, 4\,. \tag{3.18}$$

When $r = r_c^{(2)}$ we have one double root, therefore a unique fixed point.

The study of the stability of the solutions of $x = f^{(2)}(x)$ requires the same steps that led to the equation (3.15) and we determine if the absolute value of $f^{(2)\prime}(x)$ is greater, less or equal to one. By noting that[5] $f^{(2)\prime}(x_3) = f^{(2)\prime}(x_4) = f'(x_3)f'(x_4) = -r^2 + 2r + 4$, we see that for $r = r_c^{(2)} = 3$, $f^{(2)\prime}(x_3^*) = f^{(2)\prime}(x_4^2) = 1$ and for $r = r_c^{(3)} = 1 + \sqrt{6} \approx 3.4495$, $f^{(2)\prime}(x_3) = f^{(2)\prime}(x_4) = -1$. For the intermediate values $3 < r < 1 + \sqrt{6}$ the derivatives $|f^{(2)\prime}(x_\alpha^*)| < 1$ for $\alpha = 3, 4$. Therefore, these points are stable solutions of $x = f^{(2)}(x)$ and the points $x_1^*, x_2^*$ bifurcate to $x_\alpha^*$, $\alpha = 1, 2, 3, 4$ for $r = r_c^{(2)} = 3$. Almost all trajectories with initial points in the interval $[0, 1]$ are attracted by the periodic trajectory with period 2, the "2-cycle" $\{x_3^*, x_4^*\}$.

[4]Because, if $x^* = f(x^*) \Rightarrow f^{(2)}(x^*) = f(f(x^*)) = f(x^*) = x^*$ etc, the point $x^*$ is also a solution of $x^* = f^{(n)}(x^*)$.

[5]The chain rule $dh(g(x))/dx = h'(g(x))g'(x)$ gives that $f^{(2)\prime}(x_3^*) = df(f(x_3^*))/dx = f'(f(x_3^*))f'(x_3^*) = f'(x_4^*)f'(x_3^*)$ and similarly for $f^{(2)\prime}(x_4^*)$. We can prove by induction that for the $n$ solutions $x_{n+1}^*, x_{n+2}^*, \ldots, x_{2n}^*$ that belong to the $n$-cycle of the equation $x = f^{(n)}(x)$ we have that $f^{(n)\prime}(x_{n+i}) = f'(x_{n+1})\ f'(x_{n+2}) \ldots f'(x_{2n})$ for every $i = 1, \ldots, n$.

Using similar arguments we find that the fixed points $x^*_\alpha$, $\alpha = 1, 2, 3, 4$ bifurcate to the eight fixed points $x^*_\alpha$, $\alpha = 1, \ldots, 8$ when $r = r_c^{(3)} = 1 + \sqrt{6}$. These are real solutions of the equation that gives the 4-cycle $x = f^{(4)}(x)$. For $r_c^{(2)} < r < r_c^{(3)} \approx 3.5441$, the points $x^*_\alpha$, $\alpha = 5, \ldots, 8$ are a stable 4-cycle which is an attractor of almost all trajectories of the logistic map[6]. Similarly, for $r_c^{(4)} < r < r_c^{(5)}$ the 16 fixed points of the equation $x = f^{(8)}(x)$ give a stable 8-cycle, for $r_c^{(5)} < r < r_c^{(6)}$ a stable 16-cycle etc[7]. This is the phenomenon which is called *period doubling* which continues ad infinitum. The points $r_c^{(n)}$ are getting closer to each other as $n$ increases so that $\lim_{n\to\infty} r_c^{(n)} = r_c \approx 3.56994567$. As we will see, $r_c$ marks the onset of the non-periodic, chaotic behavior of the trajectories of the logistic map.

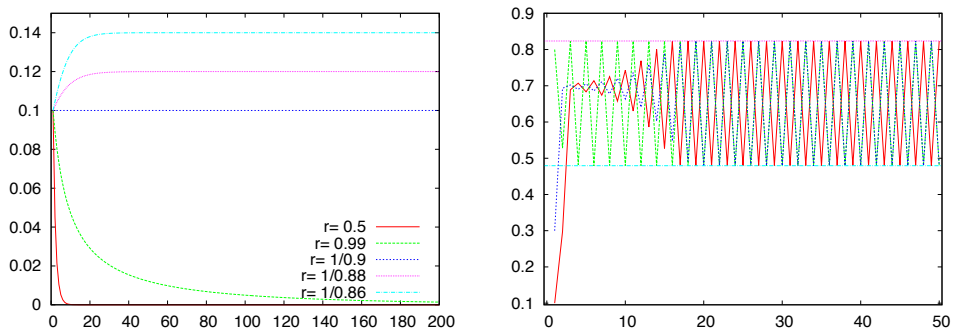



Figure 3.1: (Left) Some trajectories of the logistic map with $x_0 = 0.1$ and various values of $r$. We can see the first bifurcation for $r_c^{(1)} = 1$ from $x_1^* = 0$ to $x_2^* = 1 - 1/r$. (Right) Trajectories of the logistic map for $r_c^{(2)} < r = 3.5 < r_c^{(3)}$. The three curves start from three different initial points. After a transient period, depending on the initial point, one obtains a periodic trajectory which is a 2-cycle. The horizontal lines are the expected values $x^*_{3,4} = ((r+1) \mp \sqrt{r^2 - 2r - 3})/(2r)$ (see text).

Computing the bifurcation points becomes quickly intractable and we have to resort to a numerical computation of their values. Initially we will write a program that computes trajectories of the logistic map for chosen values of $r$ and $x_0$. The program can be found in the file `logistic.f90` and is listed below:

[6]The points $x^*_\alpha$, $\alpha = 1, \ldots, 4$ are unstable fixed points and 2-cycle.

[7]Generally, for $r_c^{(n)} < r < r_c^{(n+1)} < r_c \approx 3.56994567$ we have $2^n$ fixed points of the equation $x = f^{(2^{n-1})}(x)$ and stable $2^{n-1}$-cycles, which are attractors of almost all trajectories.

```
!==========================================================
!Discrete Logistic Map
!==========================================================
program logistic_map
 implicit none
 integer :: NSTEPS,i
 real(8) :: r,x0,x1
! ------ Input:
 print *,'# Enter NSTEPS, r, x0:'
 read  *,NSTEPS,r,x0
 print *,'# NSTEPS = ',NSTEPS
 print *,'# r      = ',r
 print *,'# x0     = ',x0
! ------ Initialize:
 open(unit=33,file='log.dat')
 write(33,*) 1,x0
! ------ Calculate:
 do i=1,NSTEPS
  x1 = r * x0 * (1.0D0-x0)
  write(33,*)i,x1
  x0 = x1
 enddo
 close(33)
end program logistic_map
```

The program is compiled and run using the commands:

```
> gfortran logistic.f90 -o l
> echo "100 0.5 0.1" | ./l
```

The command `echo` prints to the `stdout` the values of the parameters `NSTEPS=100, r=0.5` and `x0=0.1`. Its `stdout` is redirected to the `stdin` of the command `./l` by using a pipe via the symbol `|`, from which the program reads their value and uses them in the calculation. The results can be found in two columns in the file `log.dat` and can be plotted using `gnuplot`. The plots are put in figure 3.1 and we can see the first two bifurcations when $r$ goes past the values $r_c^{(1)}$ and $r_c^{(2)}$. Similarly, we can study trajectories which are $2^n$-cycles when $r$ crosses the values $r_c^{(n-1)}$.

Another way to depict the 2-cycles is by constructing the cobweb plots: We start from the point $(x_0, 0)$ and we calculate the point $(x_0, x_1)$, where $x_1 = f(x_0)$. This point belongs on the curve $y = f(x)$. The point $(x_0, x_1)$ is then projected on the diagonal $y = x$ and we obtain the point $(x_1, x_1)$. We repeat $n$ times obtaining the points $(x_n, x_{n+1})$ and $(x_{n+1}, x_{n+1})$ on $y = f(x)$

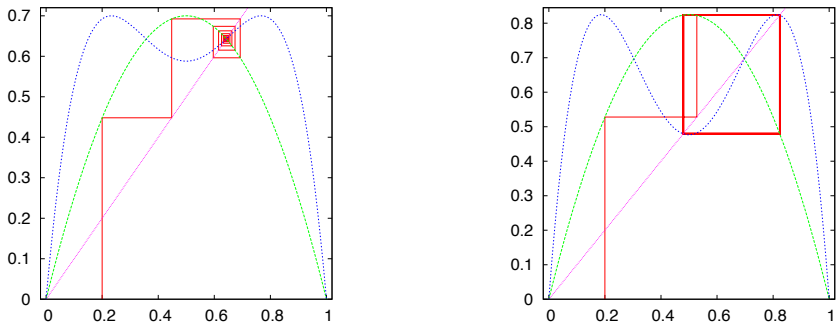



Figure 3.2: Cobweb plots of the logistic map for $r = 2.8$ and 3.3. (Left) The left plot is an example of a fixed point $x^* = f(x^*)$. The green line is $y = f(x)$ and the blue line is $y = f^{(2)}(x)$. The trajectory ends at the unique non zero intersection of the diagonal and $y = f(x)$ which is $x_2^* = 1 - 1/r$. The trajectory intersects the curve $y = f^{(2)}(x)$ at the same point. $y = f^{(2)}(x)$ does not intersect the diagonal anywhere else. (Right) The right plot shows an example of a 2-cycle. $y = f^{(2)}(x)$ intersects the diagonal at two additional points determined by $x_3^*$ and $x_4^*$. The trajectory ends up on the orthogonal $(x_3^*, x_3^*)$, $(x_4^*, x_3^*)$, $(x_4^*, x_4^*)$, $(x_3^*, x_4^*)$.

and $y = x$ respectively. The fixed points $x^* = f(x^*)$ are at the intersections of these curves and, if they are attractors, the trajectories will converge on them. If we have a $2^n$-cycle, we will observe a periodic trajectory going through points which are solutions to the equation $x = f^{(2^n)}(x)$. This exercise can be done by using the following program, which can be found in the file `logistic1.f90`:

```
!==========================================================
! Discrete Logistic Map
! Map the trajectory in 2d space (plane)
!==========================================================
program logistic_map
 implicit none
 integer :: NSTEPS,i
 real(8) :: r,x0,x1
! ------ Input:
 print *,'# Enter NSTEPS, r, x0:'
 read  *,NSTEPS,r,x0
 print *,'# NSTEPS = ',NSTEPS
 print *,'# r      = ',r
 print *,'# x0     = ',x0
! ------ Initialize:
 open(unit=33,file='trj.dat')
! ------ Calculate:
```

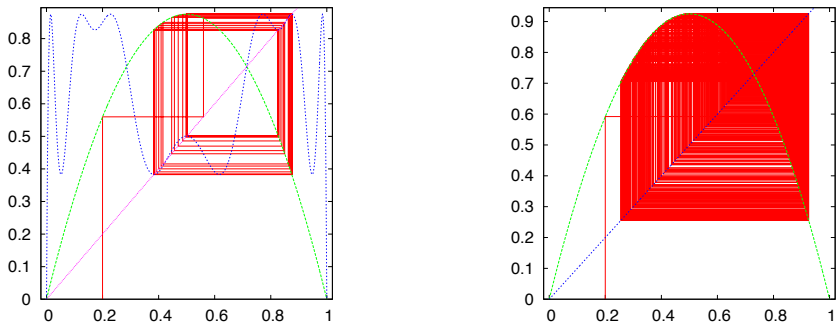



Figure 3.3: (Left) A 4-cycle for $r = 3.5$. The blue curve is $y = f^{(4)}(x)$ which intersects the diagonal at four points determined by $x_\alpha$, $\alpha = 5, 6, 7, 8$. The four cycle passes through these points. (Right) a non periodic orbit for $r = 3.7$ when the system exhibits chaotic behavior.

```
 write(33,*) 0, x0,0
 do i=1,NSTEPS
  x1 = r * x0 * (1.0D0-x0)
  write(33,*) 2*i-3,x0,x1
  write(33,*) 2*i-2,x1,x1
  x0 = x1
 enddo
 close(33)
end program logistic_map
```

Compiling and running this program is done exactly as in the case of the program in `logistic.f90`. We can plot the results using `gnuplot`. The plot in figure 3.2 can be constructed using the commands:

```
gnuplot> set size square
gnuplot> f(x) = r*x*(1.0-x)
gnuplot> r = 3.3
gnuplot> plot "<echo 50 3.3 0.2|./l;cat trj.dat" using 2:3 w l
gnuplot> replot f(x) ,f(f(x)),x
```

The `plot` command shown above, runs the program exactly as it is done on the command line. This is accomplished by using the symbol <, which reads the plot from the `stdout` of the command `"echo 50 3.3 0.2|./l;cat trj.dat"`. Only the second command `"echo trj.dat"` writes to the `stdout`, therefore the plot is constructed from the contents of the file `trj.dat`. The following line adds the plots of the functions $f(x)$, $f^{(2)}(x) = f(f(x))$ and of the diagonal $y = x$. Figures 3.2 and 3.3 show examples of attractors which are fixed points, 2-cycles and 4-cycles. An

example of a non periodic trajectory is also shown, which exhibits chaotic behavior which can happen when $r > r_c \approx 3.56994567$.

## 3.3 Bifurcation Diagrams

The bifurcations of the fixed points of the logistic map discussed in the previous section can be conveniently shown on the "bifurcation diagram". We remind to the reader that the first bifurcations happen at the critical values of r

$$r_c^{(1)} < r_c^{(2)} < r_c^{(3)} < \ldots < r_c^{(n)} < \ldots < r_c\,, \tag{3.19}$$

where $r_c^{(1)} = 1$, $r_c^{(2)} = 3$, $r_c^{(3)} = 1 + \sqrt{6}$ and $r_c = \lim_{n\to\infty} r_c^{(n)} \approx 3.56994567$. For $r_c^{(n)} < r < r_c^{(n+1)}$ we have $2^n$ fixed points $x_\alpha^*$, $\alpha = 1, 2, ..., 2^n$. By plotting these points $x_\alpha^*(r)$ as a function of $r$ we construct the bifurcation diagram. These can be calculated numerically by using the program `bifurcate.f90`. In this program, the user selects the values of $r$ that she needs to study and for each one of them the program records the point of the $2^{n-1}$-cycles[8] $x_\alpha^*(r)$, $\alpha = 2^{n-1}+1, 2^{n-1}+2, \ldots, 2^n$. This is easily done by computing the logistic map several times until we are sure that the trajectories reach the stable state. The parameter `NTRANS` in the program determines the number of points that we throw away, which should contain all the *transient* behavior. After `NTRANS` steps, the program records `NSTEPS` points, where `NSTEPS` should be large enough to cover all the points of the $2^{n-1}$-cycles or depict a dense enough set of values of the non periodic orbits. The program is listed below:

```
!==========================================================
! Bifurcation Diagram of the Logistic Map
!==========================================================
program bifurcation_diagram
 implicit none
 real(8),parameter :: rmin   = 2.5D0
 real(8),parameter :: rmax   = 4.0D0
 integer,parameter :: NTRANS = 500   !Number of discarded steps
 integer,parameter :: NSTEPS = 100   !Number of recorded  steps
 integer,parameter :: RSTEPS = 2000  !Number of values of r
 integer           :: i
 real(8)           :: r,dr,x0,x1
! ------ Initialize:
```

[8] If we want to be more precise, the bifurcation diagram contains also the unstable points. What we really construct is the orbit diagram which contains only the stable points.

```
 open(unit=33,file='bif.dat')
 dr      = (rmax-rmin)/RSTEPS !Increment in r
! ----- Calculate:
 r = rmin
 do while ( r .le. rmax)
  x0 = 0.5D0
! ---- Transient steps: skip
  do i=1,NTRANS
   x1 = r * x0 * (1.0D0-x0)
   x0 = x1
  enddo
  do i=1,NSTEPS
   x1 = r * x0 * (1.0D0-x0)
   write(33,*) r,x1
   x0 = x1
  enddo
  r = r + dr
 enddo ! do while
 close(33)
end program bifurcation_diagram
```

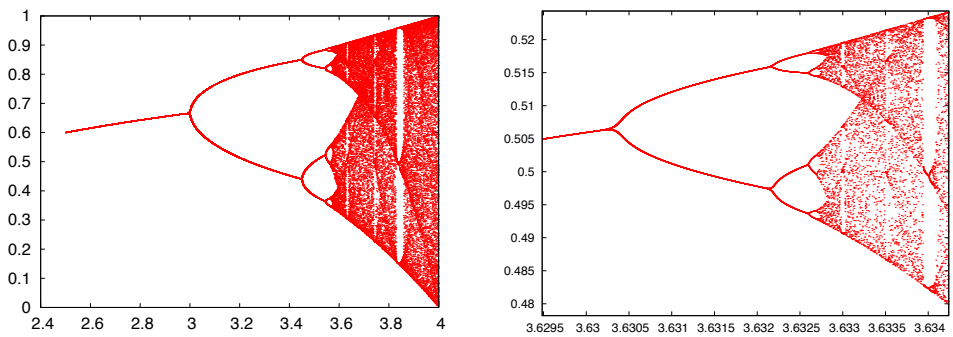



Figure 3.4: (Left) The bifurcation diagram computed by the program `bifurcate.f90` for $2.5 < r < 4$. Notice the first bifurcation points followed by intervals of chaotic, non-periodic orbits interrupted by intermissions of stable periodic trajectories. The chaotic trajectories take values in subsets of the interval $(0, 1)$. For $r = 4$ they take values within the whole $(0, 1)$. One can see that for $r = 1 + \sqrt{8} \approx 3.8284$ we obtain a 3-cycle which subsequently bifurcates to $3 \cdot 2^n$-cycles. (Right) The diagram on the left is magnified in a range of $r$ showing the self-similarity of the diagram at all scales.

The program can be compiled and run using the commands:

```
> gfortran bifurcate.f90 -o b
> ./b;
```

The left plot of figure 3.4 can be constructed by the `gnuplot` commands:

```
gnuplot> plot "bif.dat" with dots
```

We observe the fixed points and the $2^n$-cycles for $r < r_c$. When $r$ goes past $r_c$, the trajectories become non-periodic and exhibit chaotic behavior. Chaotic behavior will be discussed more extensively in the next section. For the time being, we note that if we measure the distance between the points $\Delta r^{(n)} = r_c^{(n+1)} - r_c^{(n)}$, we find that it decreases constantly with $n$ so that

$$\lim_{n\to\infty} \frac{\Delta r^{(n)}}{\Delta r^{(n+1)}} = \delta \approx 4.669\,201\,609\,, \tag{3.20}$$

where $\delta$ is the Feigenbaum constant. An additional constant $\alpha$, defined by the quotient of the separation of adjacent elements $\Delta w_n$ of period doubled attractors from one double to the next $\Delta w_{n+1}$, is

$$\lim_{n\to\infty} \frac{\Delta w_n}{\Delta w_{n+1}} = \alpha \approx 2.502\,907\,875\,. \tag{3.21}$$

It is also interesting to note the appearance of a 3-cycle right after $r = 1 + \sqrt{8} \approx 3.8284 > r_c$! By using the theorem of Sharkovskii, Li and Yorke[9] showed that any one dimensional system has 3-cycles, therefore it will have cycles of any length and chaotic trajectories. The stability of the 3-cycle can be studied from the solutions of $x = f^{(3)}(x)$ in exactly the same way that we did in equations (3.16) and (3.17) (see [19] for details). Figure 3.5 magnifies a branch of the 3-cycle. By magnifying different regions in the bifurcation plot, as shown in the right plot of figure 3.4, we find similar shapes to the branching of the 3-cycle. Figure 3.4 shows that between intervals of chaotic behavior we obtain "windows" of periodic trajectories. These are infinite but countable. It is also quite interesting to note that if we magnify a branch withing these windows, we obtain a diagram that is similar to the whole diagram! We say that the bifurcation diagram exhibits self similarity. There are more interesting properties of the bifurcation diagram and we refer the reader to the bibliography for a more complete exposition.

We close this section by mentioning that the qualitative properties of the bifurcation diagram are the same for a whole class of functions. Feigenbaum discovered that if one takes any function that is concave and

[9] T.Y. Li, J.A. Yorke, "Period Three Implies Chaos", American Mathematical Monthly 82 (1975) 985.

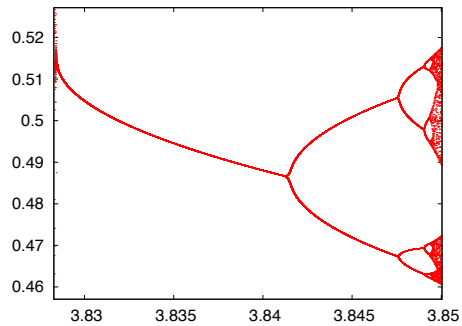



Figure 3.5: Magnification of one of the three branches of the 3-cycle for $r > 1 + \sqrt{8}$. To the left, we observe the temporary halt of the chaotic behavior of the trajectory, which comes back as shown in the plot to the right after an intermission of stable periodic trajectories.

has a unique global maximum, its bifurcation diagram behaves qualitatively the same way as that of the logistic map. Examples of such functions[10] studied in the literature are $g(x) = x\mathrm{e}^{r(1-x)}$, $u(x) = r\sin(\pi x)$ and $w(x) = b - x^2$. The constants $\delta$ and $\alpha$ of equations (3.20) and (3.21) are the same of all these mappings. The functions that result in chaotic behavior are studied extensively in the literature and you can find a list of those in [25].

## 3.4 The Newton-Raphson Method

In order to determine the bifurcation points, one has to solve the nonlinear, polynomial, algebraic equations $x = f^{(n)}(x)$ and $f^{(n)\prime}(x) = -1$. For this reason, one has to use an approximate numerical calculation of the roots, and the simple Newton-Raphson method will prove to be a good choice.

Newton-Raphson's method uses an initial guess $x_0$ for the solution of the equation $g(x) = 0$ and computes a sequence of points $x_1, x_2, \ldots, x_n, x_{n+1}, \ldots$ that presumably converges to one of the roots of the equation. The computation stops at a finite $n$, when we decide that the desired level of accuracy has been achieved. In order to understand how it works, we assume that $g(x)$ is an analytic function for all the values of $x$ used in

[10] The function $x\exp(r(1-x))$ has been used as a model for populations whose large density is restricted by epidemics. The populations are always positive independently of the (positive) initial conditions and the value of $r$.

the computation. Then, by Taylor expanding around $x_n$ we obtain

$$g(x_{n+1}) = g(x_n) + (x_{n+1} - x_n)g'(x) + \dots . \tag{3.22}$$

If we wish to have $g(x_{n+1}) \approx 0$, we choose

$$x_{n+1} = x_n - \frac{g(x_n)}{g'(x_n)} . \tag{3.23}$$

The equation above gives the Newton-Raphson method for one equation $g(x) = 0$ of one variable $x$. Different choices for $x_0$ will possibly lead to different roots. When $g'(x)$, $g''(x)$ are non zero at the root and $g'''(x)$ is bounded, the convergence of the method is quadratic with the number of iterations. This means that there is a neighborhood of the root $\alpha$ such that the distance $\Delta x_{n+1} = x_{n+1} - \alpha$ is $\Delta x_{n+1} \propto (\Delta x_n)^2$. If the root $\alpha$ has multiplicity larger than 1, convergence is slower. The proofs of these statements are simple and can be found in [26].

The Newton-Raphson method is simple to program and, most of the times, sufficient for the solution of many problems. In the general case it works well only close enough to a root. We should also keep in mind that there are simple reasons for the method to fail. For example, when $g'(x_n) = 0$ for some $n$, the method stops. For functions that tend to 0 as $x \to \pm\infty$, it is easy to make a bad choice for $x_0$ that does not lead to convergence to a root. Sometimes it is a good idea to combine the Newton-Raphson method with the *bisection* method. When the derivative $g'(x)$ diverges at the root we might get into trouble. For example, the equation $|x|^\nu = 0$ with $0 < \nu < 1/2$, does not lead to a convergent sequence. In some cases, we might enter into non-convergent cycles [8]. For some functions the basin of attraction of a root (the values of $x_0$ that will converge to the root) can be tiny. See problem 13.

As a test case of our program, consider the equation

$$\epsilon \tan \epsilon = \sqrt{\rho^2 - \epsilon^2} \tag{3.24}$$

which results from the solution of Schrödinger's equation for the energy spectrum of a quantum mechanical particle of mass $m$ in a one dimensional potential well of depth $V_0$ and width $L$. The parameters $\epsilon = \sqrt{mL^2E/(2\hbar)}$ και $\rho = \sqrt{mL^2V_0/(2\hbar)}$. Given $\rho$, we solve for $\epsilon$ which gives the energy $E$. The function $g(x)$ and its derivative $g'(x)$ are

$$\begin{aligned} g(x) &= x \tan x - \sqrt{\rho^2 - x^2} \\ g'(x) &= \frac{x}{\sqrt{\rho^2 - x^2}} + \frac{x}{\cos^2 x} + \tan x . \end{aligned} \tag{3.25}$$

The program of the Newton-Raphson method for solving the equation $g(x) = 0$ can be found in the file `nr.f90`:

```
!==========================================================
!Newton Raphson for a function of one variable
!==========================================================
program NewtonRaphson
 implicit none
 real(8), parameter :: rho  = 15.0D0
 real(8), parameter :: eps  = 1D-6
 integer, parameter :: NMAX = 1000
 real(8) :: x0, x1, err, g, gp
 integer :: i
 print *, 'Enter x0: '
 read  *, x0
 err = 1.0D0
 print *,'iter          x                          error    '
 print *,'-------------------------------------------------'
 print *, 0,x0,err
 do i=1,NMAX
!value of function g(x):
  g   = x0*tan(x0)-sqrt(rho*rho-x0*x0)
!value of the derivative g'(x):
  gp  = x0/sqrt(rho*rho-x0*x0)+x0/(cos(x0)**2)+tan(x0)
  x1  = x0 - g/gp
  err = ABS(x1-x0)
  print *,i,x1,err
  if(err .lt. eps) exit
  x0 = x1
 enddo
end program NewtonRaphson
```

In the program listed above, the user is asked to set the initial point $x_0$. We fix $\rho =$ `rho` $= 15$. It is instructive to make the plot of the left and right hand sides of (3.24) and make a graphical determination of the roots from their intersections. Then we can make appropriate choices of the initial point $x_0$. Using `gnuplot`, the plots are made with the commands:

```
gnuplot> g1(x) = x*tan(x)
gnuplot> g2(x) = sqrt(rho*rho-x*x)
gnuplot> plot [0:20][0:20]  g1(x), g2(x)
```

The compilation and running of the program can be done as follows:

```
> gfortran nr.f90 -o n
```

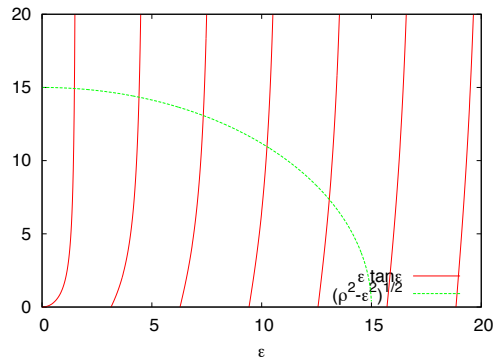



Figure 3.6: Plots of the right and left hand sides of equation (3.24). The intersections of the curves determine the solutions of the equation and their approximate graphical estimation can serve as initial points $x_0$ for the Newton-Raphson method.

```
> echo "1.4"|./n
 Enter x0:
 iter x                          error
 ------------------------------------------------------
 0     1.3999999999999999        1.0000000000000000
 1     1.5254292024457967        0.12542920244579681
 2     1.5009739120496131        2.4455290396183660E-002
 3     1.4807207017202200        2.0253210329393090E-002
 4     1.4731630533073483        7.5576484128716537E-003
 5     1.4724779331237687        6.8512018357957949E-004
 6     1.4724731072313519        4.8258924167932093E-006
 7     1.4724731069952235        2.3612845012621619E-010
```

We conclude that one of the roots of the equation is $\epsilon \approx 1.472473107$. The reader can compute more of these roots by following these steps by herself.

The method discussed above can be easily generalized to the case of two equations. Suppose that we need to solve simultaneously two algebraic equations $g_1(x_1, x_2) = 0$ and $g_2(x_1, x_2) = 0$. In order to compute a sequence $(x_{10}, x_{20})$, $(x_{11}, x_{21})$, ..., $(x_{1n}, x_{2n})$, $(x_{1(n+1)}, x_{2(n+1)})$, ... that may converge to a root of the above system of equations, we Taylor expand

the two functions around $(x_{1n}, x_{2n})$

$$\begin{aligned} g_1(x_{1(n+1)}, x_{2(n+1)}) &= g_1(x_{1n}, x_{2n}) + (x_{1(n+1)} - x_{1n})\frac{\partial g_1(x_{1n}, x_{2n})}{\partial x_1} \\ &+ (x_{2(n+1)} - x_{2n})\frac{\partial g_1(x_{1n}, x_{2n})}{\partial x_2} + \ldots \\ g_2(x_{1(n+1)}, x_{2(n+1)}) &= g_2(x_{1n}, x_{2n}) + (x_{1(n+1)} - x_{1n})\frac{\partial g_2(x_{1n}, x_{2n})}{\partial x_1} \\ &+ (x_{2(n+1)} - x_{2n})\frac{\partial g_2(x_{1n}, x_{2n})}{\partial x_2} + \ldots . \end{aligned} \tag{3.26}$$

Defining $\delta x_1 = (x_{1(n+1)} - x_{1n})$ and $\delta x_2 = (x_{2(n+1)} - x_{2n})$ and setting $g_1(x_{1(n+1)}, x_{2(n+1)}) \approx 0$, $g_2(x_{1(n+1)}, x_{2(n+1)}) \approx 0$, we obtain

$$\begin{aligned} \delta x_1 \frac{\partial g_1}{\partial x_1} + \delta x_2 \frac{\partial g_1}{\partial x_2} &= -g_1 \\ \delta x_1 \frac{\partial g_2}{\partial x_1} + \delta x_2 \frac{\partial g_2}{\partial x_2} &= -g_2 \,. \end{aligned} \tag{3.27}$$

This is a linear $2 \times 2$ system of equations

$$\begin{aligned} A_{11}\delta x_1 + A_{12}\delta x_2 &= b_1 \\ A_{21}\delta x_1 + A_{22}\delta x_2 &= b_2 \,, \end{aligned} \tag{3.28}$$

where $A_{ij} = \partial g_i / \partial x_j$ and $b_i = -g_i$, with $i, j = 1, 2$. Solving for $\delta x_i$ we obtain

$$\begin{aligned} x_{1(n+1)} &= x_{1n} + \delta x_1 \\ x_{2(n+1)} &= x_{2n} + \delta x_2 \,. \end{aligned} \tag{3.29}$$

The iterations stop when $\delta x_i$ become small enough.

As an example, consider the equations with $g_1(x) = 2x^2 - 3xy + y - 2$, $g_2(x) = 3x + xy + y - 1$. We have $A_{11} = 4x - 3y$, $A_{12} = 1 - 3x$, $A_{21} = 3 + y$, $A_{22} = 1 + x$. The program can be found in the file `nr2.f90`:

```
!=============================================================
!Newton Raphson of two functions of two variables
!=============================================================
program NewtonRaphson2
 implicit none
 real(8), parameter :: eps  = 1D-6
 integer, parameter :: NMAX = 1000
 real(8) :: A(2,2),b(2),dx(2)
```

```
 real(8) :: x,y, err
 integer :: i
 print *, 'Enter x0,y0: '
 read  *, x,y
 err = 1.0D0
 print *,'iter            x            y              error     '
 print *,'_____________________________________________________'
 print *, 0,x,y,err
 do i=1,NMAX
  b(1)   = -(2.0D0*x*x-3.0D0*x*y+y-2.0D0) ! -g1(x,y)
  b(2)   = -(3.0D0*x + x*y + y - 1.0D0)   ! -g2(x,y)
! dg1/dx                      dg1/dy
  A(1,1) = 4.0D0*x-3.0D0*y; A(1,2) = 1.0D0-3.0D0*x
! dg2/dx                      dg2/dy
  A(2,1) = 3.0D0+y         ; A(2,2) = 1.0D0+x
  call solve2x2(A,B,dx)
  x = x + dx(1)
  y = y + dx(2)
  err = 0.5D0*SQRT(dx(1)**2+dx(2)**2)
  print *,i,x,y,err
  if(err .lt. eps) exit
 enddo
end program NewtonRaphson2
!==========================================================
subroutine solve2x2(A,b,dx)
 implicit none
 real(8) :: A(2,2),b(2),dx(2)
 real(8) :: num1,num2,det
 num1 = A(2,2)*b(1)-A(1,2)*b(2)
 num2 = A(1,1)*b(2)-A(2,1)*b(1)
 det  = A(1,1)*A(2,2)-A(1,2)*A(2,1)
 if(det .eq. 0.0D0) stop 'solve2x2: det=0'
 dx(1)= num1/det
 dx(2)= num2/det
end subroutine solve2x2
```

In order to guess the region where the real roots of the systems lie, we make a 3-dimensional plot using `gnuplot`:

```
gnuplot> set isosamples 20
gnuplot> set hidden3d
gnuplot> splot 2*x**2-3*x*y+y-2,3*x+y*x+y-1,0
```

We plot the functions $g_i(x, y)$ together with the plane $x = 0$. The intersection of the three surfaces determine the roots we are looking for. Compiling and running the program can be done by using the commands:

```
> gfortran nr2.f90 -o n
> echo 2.2 1.5 |./n
 Enter x0,y0:
 iter    x          y       error
 ------------------------------------------
 0  2.20000000  1.50000000 1.0000
 1  0.76427104  0.26899383 0.9456
 2  0.73939531 -0.68668275 0.4780
 3  0.74744506 -0.71105605 1.2834E-002
 4  0.74735933 -0.71083147 1.2019E-004
 5  0.74735932 -0.71083145 1.2029E-008
> echo 0 1 |./n
..................
 5 -0.10899022  1.48928857 4.3461E-012
> echo -5 0|./n
 6 -6.13836909 -3.77845711 3.2165E-013
```

The computation above leads to the roots $(0.74735932,\ -0.71083145)$, $(-0.10899022,\ 1.48928857)$, $(-6.13836909,\ -3.77845711)$.

The Newton-Raphson method for many variables becomes hard quite soon: One needs to calculate the functions as well as their derivatives, which is prohibitively expensive for many problems. It is also hard to determine the roots, since the method converges satisfactorily only very close to the roots. We refer the reader to [8] for more information on how one can deal with these problems.

## 3.5 Calculation of the Bifurcation Points

In order to determine the bifurcation points for $r < r_c$ we will solve the algebraic equations $x = f^{(k)}(x)$ and $f^{(k)\prime}(x) = -1$. At these points, $k$-cycles become unstable and $2k$-cycles appear and are stable. This happens when $r = r_c^{(n)}$, where $k = 2^{n-2}$. We will look for solutions $(x_\alpha^*, r_c^{(n)})$ for $\alpha = k+1, k+2, \ldots, 2k$.

We define the functions $F(x,r) = f(x) = rx(1-x)$ and $F^{(k)}(x,r) = f^{(k)}(x)$ as in equation (3.6). We will solve the algebraic equations:

$$\begin{aligned} g_1(x,r) &= x - F^{(k)}(x,r) = 0 \\ g_2(x,r) &= \frac{\partial F^{(k)}(x,r)}{\partial x} + 1 = 0\,. \end{aligned} \tag{3.30}$$

According to the discussion of the previous section, in order to calculate the roots of these equations we have to solve the linear system (3.28),

where the coefficients are

$$
\begin{aligned}
b_1 &= -g_1(x,r) = -x + F^{(k)}(x,r) \\
b_2 &= -g_2(x,r) = -\frac{\partial F^{(k)}(x,r)}{\partial x} - 1 \\
A_{11} &= \frac{\partial g_1(x,r)}{\partial x} = 1 - \frac{\partial F^{(k)}(x,r)}{\partial x} \\
A_{12} &= \frac{\partial g_1(x,r)}{\partial r} = -\frac{\partial F^{(k)}(x,r)}{\partial r} \\
A_{21} &= \frac{\partial g_2(x,r)}{\partial x} = \frac{\partial^2 F^{(k)}(x,r)}{\partial x^2} \\
A_{22} &= \frac{\partial g_2(x,r)}{\partial r} = \frac{\partial^2 F^{(k)}(x,r)}{\partial x \partial r} \,. \qquad (3.31)
\end{aligned}
$$

The derivatives will be calculated approximately using finite differences

$$
\begin{aligned}
\frac{\partial F^{(k)}(x,r)}{\partial x} &\approx \frac{F^{(k)}(x+\epsilon,r) - F^{(k)}(x-\epsilon,r)}{2\epsilon} \\
\frac{\partial F^{(k)}(x,r)}{\partial r} &\approx \frac{F^{(k)}(x,r+\epsilon) - F^{(k)}(x,r-\epsilon)}{2\epsilon} \,, \qquad (3.32)
\end{aligned}
$$

and similarly for the second derivatives

$$
\begin{aligned}
\frac{\partial^2 F^{(k)}(x,r)}{\partial x^2} &\approx \frac{\frac{\partial F^{(k)}(x+\frac{\epsilon}{2},r)}{\partial x} - \frac{\partial F^{(k)}(x-\frac{\epsilon}{2},r)}{\partial x}}{2\frac{\epsilon}{2}} \\
&= \frac{1}{\epsilon}\left\{\frac{F^{(k)}(x+\epsilon,r) - F^{(k)}(x,r)}{\epsilon} - \frac{F^{(k)}(x,r) - F^{(k)}(x-\epsilon,r)}{\epsilon}\right\} \\
&= \frac{1}{\epsilon^2}\left\{F^{(k)}(x+\epsilon,r) - 2F^{(k)}(x,r) + F^{(k)}(x-\epsilon,r)\right\} \\
\frac{\partial^2 F^{(k)}(x,r)}{\partial x \partial r} &\approx \frac{\frac{\partial F^{(k)}(x+\epsilon_x,r)}{\partial r} - \frac{\partial F^{(k)}(x-\epsilon_x,r)}{\partial r}}{2\epsilon_x} \\
&= \frac{1}{2\epsilon_x}\left\{\frac{F^{(k)}(x+\epsilon_x,r+\epsilon_r) - F^{(k)}(x+\epsilon_x,r-\epsilon_r)}{2\epsilon_r}\right. \\
&\qquad \left. - \frac{F^{(k)}(x-\epsilon_x,r+\epsilon_r) - F^{(k)}(x-\epsilon_x,r-\epsilon_r)}{2\epsilon_r}\right\} \\
&= \frac{1}{4\epsilon_x\epsilon_r}\left\{F^{(k)}(x+\epsilon_x,r+\epsilon_r) - F^{(k)}(x+\epsilon_x,r-\epsilon_r)\right. \\
&\qquad \left. - F^{(k)}(x-\epsilon_x,r+\epsilon_r) + F^{(k)}(x-\epsilon_x,r-\epsilon_r)\right\} \qquad (3.33)
\end{aligned}
$$

We are now ready to write the program for the Newton-Raphson method like in the previous section. The only difference is the approximate calculation of the derivatives using the relations above and the calculation

of the function $F^{(k)}(x,r)$ by a routine that will compose the function $f(x)$ $k$-times. The program can be found in the file `bifurcationPoints.f90`:

```
!==========================================================
!        bifurcationPoints.f
! Calculate bifurcation points of the discrete logistic map
! at period k by solving the condition
! g1(x,r) = x - F(k,x,r)    = 0
! g2(x,r) = dF(k,x,r)/dx+1 = 0
! determining when the Floquet multiplier becomes 1
! F(k,x,r) iterates F(x,r) = r*x*(x-1) k times
! The equations are solved by using a Newton-Raphson method
!==========================================================
program bifurcationPoints
 implicit none
 real(8),parameter :: tol=1.0D-10
 integer :: k,iter
 real(8) :: r0,x0
 real(8) :: A(2,2),B(2),dX(2)
 real(8) :: error
 real(8) :: F,dFdx,dFdr,d2Fdx2,d2Fdrdx
! ---- Input:
 print *,'# Enter k,r0,x0:'
 read  *,k,r0,x0
 print *,'# Period k= ',k
 print *,'# r0= ',r0,' x0= ',x0
! ---- Initialize
 error   = 1.0D0          !initial large value of error>tol
 iter    = 0
 do while(error .gt. tol)
! ---- Calculate jacobian matrix
  A(1,1) = 1.0D0-dFdx(k,x0,r0)
  A(1,2) = -dFdr     (k,x0,r0)
  A(2,1) = d2Fdx2    (k,x0,r0)
  A(2,2) = d2Fdrdx   (k,x0,r0)
  B(1)   = -x0 +    F(k,x0,r0)
  B(2)   = -dFdx     (k,x0,r0)-1.0D0
! ---- Solve a 2x2 linear system:
  call solve2x2(A,B,dX)
  x0     = x0 + dX(1)
  r0     = r0 + dX(2)
  error  = 0.5D0*sqrt(dX(1)**2+dX(2)**2)
  iter   = iter+1
  print*,iter,'x0= ',x0,' r0= ',r0,' err=',error
 enddo !do while(error .gt. tol)
end program bifurcationPoints
!==========================================================
!Function F(k,x,r) and its derivatives
```

```
real(8) function F(k,x,r)
 implicit none
 real(8) :: x,r,x0
 integer k,i

 x0  = x
 do i=1,k
  x0 = r*x0*(1.0D0-x0)
 enddo
 F   = x0

end function F
! ------------------------------------
real(8) function dFdx(k,x,r)
 implicit none
 real(8) :: x,r,eps
 real(8) :: F
 integer k

 eps     = 1.0D-6*x
 dFdx    = (F(k,x+eps,r)-F(k,x-eps,r))/(2.0D0*eps)
end function dFdx
! ------------------------------------
real(8) function dFdr(k,x,r)
 implicit none
 real(8) :: x,r,eps
 real(8) :: F
 integer k

 eps     = 1.0D-6*r
 dFdr    = (F(k,x,r+eps)-F(k,x,r-eps))/(2.0D0*eps)
end function dFdr
! ------------------------------------
real(8) function d2Fdx2(k,x,r)
 implicit none
 real(8) :: x,r,eps
 real(8) :: F
 integer k

 eps     = 1.0D-6*x
 d2Fdx2  = (F(k,x+eps,r)-2.0D0*F(k,x,r)+F(k,x-eps,r))/(eps*eps)
end function d2Fdx2
! ------------------------------------
real(8) function d2Fdrdx(k,x,r)
 implicit none
 real(8) :: x,r,epsx,epsr
 real(8) :: F
 integer k
```

```
 epsx    = 1.0D-6*x
 epsr    = 1.0D-6*r
 d2Fdrdx = (F(k,x+epsx,r+epsr)-F(k,x+epsx,r-epsr) &
      -F(k,x-epsx,r+epsr)+F(k,x-epsx,r-epsr))     &
          /(4.0D0*epsx*epsr)
end function d2Fdrdx
!==========================================================
subroutine solve2x2(A,b,dx)
 implicit none
 real(8) :: A(2,2),b(2),dx(2)
 real(8) :: num1,num2,det
 num1 = A(2,2)*b(1)  - A(1,2)*b(2)
 num2 = A(1,1)*b(2)  - A(2,1)*b(1)
 det  = A(1,1)*A(2,2)- A(1,2)*A(2,1)
 if(det .eq. 0.0D0) stop 'solve2x2: det = 0'
 dx(1) = num1/det
 dx(2) = num2/det
end subroutine solve2x2
```

Compiling and running the program can be done as follows:

```
> gfortran bifurcationPoints.f90 -o b
> echo 2 3.5 0.5 |./b
 # Enter k,r0,x0:
 # Period k=            2
 # r0= 3.5000000000000 x0= 0.50000000000
 1 x0= 0.4455758353187 r0= 3.38523275827 err= 6.35088E-002
 2 x0= 0.4396562547624 r0= 3.45290970406 err= 3.39676E-002
 3 x0= 0.4399593001407 r0= 3.44949859951 err= 1.71226E-003
 4 x0= 0.4399601690333 r0= 3.44948974267 err= 4.44967E-006
 5 x0= 0.4399601689937 r0= 3.44948974281 err= 7.22160E-011
> echo 2 3.5 0.85 | ./b
 ................
 4 x0= 0.8499377795512 r0= 3.44948974275 err= 1.85082E-011
> echo 4 3.5 0.5 |./b
 ................
 5 x0= 0.5235947861540 r0= 3.54409035953 err= 1.86318E-011
> echo 4 3.5 0.35 | ./b
 ................
 5 x0= 0.3632903374118 r0= 3.54409035955 err= 5.91653E-013
```

The above listing shows the points of the 2-cycle and some of the points of the 4-cycle. It is also possible to compare the calculated value $r_c^{(3)} = 3.449490132$ with the expected one $r_c^{(3)} = 1+\sqrt{6} \approx 3.449489742$. Improving the accuracy of the calculation is left as an exercise for the reader who has to control the systematic errors of the calculations and achieve better accuracy in the computation of $r_c^{(4)}$.

## 3.6 Liapunov Exponents

We have seen that when $r > r_c \approx 3.56994567$, the trajectories of the logistic map become non periodic and exhibit chaotic behavior. Chaotic behavior mostly means sensitivity of the evolution of a dynamical system to the choice of initial conditions. More precisely, it means that two different trajectories constructed from infinitesimally close initial conditions, diverge very fast from each other. This implies that there is a set of initial conditions of nonzero measure that do not approach arbitrarily close to any cycle of finite length.

Assume that two trajectories have $x_0$, $\tilde{x}_0$ as initial points and $\Delta x_0 = x_0 - \tilde{x}_0$. When the points $x_n$, $\tilde{x}_n$ have a distance $\Delta x_n = \tilde{x}_n - x_n$ that for small enough $n$ increases exponentially with $n$ (the "time"), i.e.

$$\Delta x_n \sim \Delta x_0 \mathrm{e}^{\lambda n}\,, \qquad \lambda > 0\,, \tag{3.34}$$

the system is most likely exhibiting chaotic behavior[11]. The exponent $\lambda$ is called a *Liapunov exponent*. A useful equation for the calculation of $\lambda$ is

$$\lambda = \lim_{n\to\infty} \frac{1}{n} \sum_{k=0}^{n-1} \ln |f'(x_k)|\,. \tag{3.35}$$

This relation can be easily proved by considering infinitesimal $\epsilon \equiv |\Delta x_0|$

[11]Sensitivity to the initial condition alone does not necessarily imply chaos. It is necessary to have topological mixing and dense periodic orbits. Topological mixing means that every open set in phase space will evolve to a set that for large enough time will have non zero intersection with any open set. Dense periodic orbits means that every point in phase space lies infinitesimally close to a periodic orbit.

so that $\lambda = \lim_{n\to\infty}\lim_{\epsilon\to 0}|\Delta x_n|/\epsilon$. Then we obtain

$$
\begin{aligned}
\tilde{x}_1 &= f(\tilde{x}_0) = f(x_0+\epsilon) \approx f(x_0) + \epsilon f'(x_0) \\
&= x_1 + \epsilon f'(x_0) \Rightarrow \\
\frac{\Delta x_1}{\epsilon} &= \frac{\tilde{x}_1 - x_1}{\epsilon} \approx f'(x_0) \\
\\
\tilde{x}_2 &= f(\tilde{x}_1) = f(x_0+\epsilon f'(x_0)) \approx f(x_1) + (\epsilon f'(x_0))f'(x_1) \\
&= x_2 + \epsilon f'(x_0)f'(x_1) \Rightarrow \\
\frac{\Delta x_2}{\epsilon} &= \frac{\tilde{x}_2 - x_2}{\epsilon} \approx f'(x_0)f'(x_1) \\
\\
\tilde{x}_3 &= f(\tilde{x}_3) = f(x_2+\epsilon f'(x_0)f'(x_1)) \approx f(x_2) + (\epsilon f'(x_0)f'(x_1))f'(x_2) \\
&= x_3 + \epsilon f'(x_0)f'(x_1)f'(x_2) \Rightarrow \\
\frac{\Delta x_3}{\epsilon} &= \frac{\tilde{x}_3 - x_3}{\epsilon} \approx f'(x_0)f'(x_1)f'(x_2)\,. \qquad (3.36)
\end{aligned}
$$

We can show by induction that $|\Delta x_n|/\epsilon \approx f'(x_0)f'(x_1)f'(x_2)\ldots f'(x_{n-1})$ and by taking the logarithm and the limits we can prove (3.35).

A first attempt to calculate the Liapunov exponents could be made by using the definition (3.34). We modify the program `logistic.f90` so that it calculates two trajectories whose initial distance is $\epsilon =$ `epsilon`:

```
!==========================================================
!Discrete Logistic Map:
!Two trajectories with close initial conditions.
!==========================================================
program logistic_map
 implicit none
 integer :: NSTEPS,i
 real(8) :: r,x0,x1,x0t,x1t,epsilon

! ------ Input:
 print *,'# Enter NSTEPS, r, x0, epsilon:'
 read  *,NSTEPS,r,x0,epsilon
 print *,'# NSTEPS  = ',NSTEPS
 print *,'# r       = ',r
 print *,'# x0      = ',x0
 print *,'# epsilon = ',epsilon

 x0t = x0+epsilon
! ------ Initialize:
 open(unit=33,file='lia.dat')
 write(33,*) 1,x0,x0t,ABS(x0t-x0)/epsilon
```

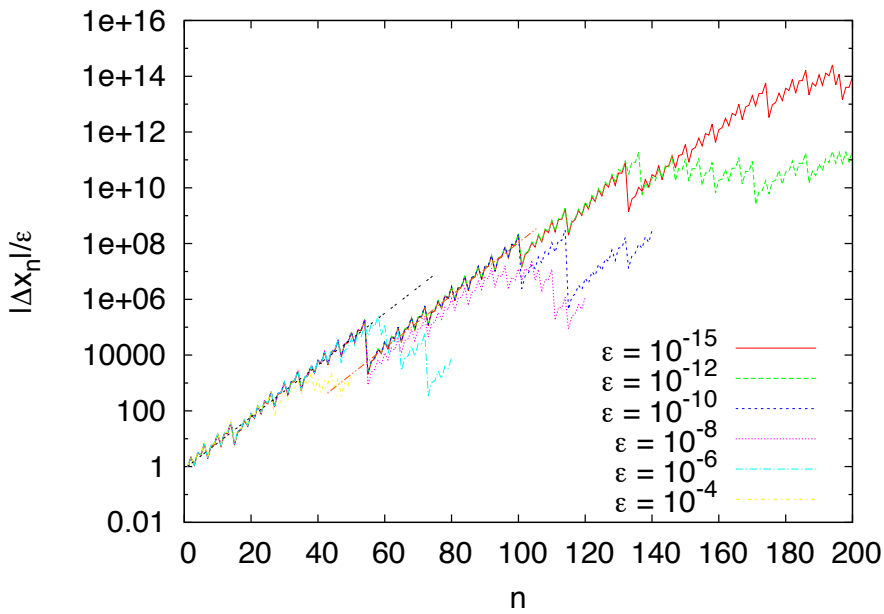



Figure 3.7: A plot of $|\Delta x_n|/\epsilon$ for the logistic map for $r = 3.6$, $x_0 = 0.2$. Note the convergence of the curves as $\epsilon \to 0$ and the approximate exponential behavior in this limit. The two lines are fits to the equation (3.34) and give $\lambda = 0.213(4)$ and $\lambda = 0.217(6)$ respectively.

```
! ------ Calculate:
 do i=2,NSTEPS
  x1  = r * x0  * (1.0D0-x0 )
  x1t = r * x0t * (1.0D0-x0t)
  write(33,*)i,x1,x1t,ABS(x1t-x1)/epsilon
  x0  = x1; x0t = x1t
 enddo
 close(33)
end program logistic_map
```

After running the program, the quantity $|\Delta x_n|/\epsilon$ is found at the fourth column of the file `lia.dat`. The curves of figure 3.7 can be constructed by using the commands:

```
> gfortran liapunov1.f90 -o l
> gnuplot
gnuplot> set logscale y
gnuplot> plot \
  "<echo 200 3.6 0.2 1e-15 |./l;cat lia.dat" u 1:4 w l
```

The last line plots the `stdout` of the command `"echo 200 3.6 0.2 1e-15 |./l;cat lia.dat"`, i.e. the contents of the file `lia.dat` produced after running our program using the parameters `NSTEPS` $= 200$, `r` $= 3.6$, `x0` $= 0.2$ και `epsilon` $= 10^{-15}$. The `gnuplot` command `set logscale y`, puts the y axis in a logarithmic scale. Therefore an exponential function is shown as a straight line and this is what we see in figure 3.7: The points $|\Delta x_n|/\epsilon$ tend to lie on a straight line as $\epsilon$ decreases. The slopes of these lines are equal to the Liapunov exponent $\lambda$. Deviations from the straight line behavior indicates corrections and systematic errors, as we point out in figure 3.7. A different initial condition results in a slightly different value of $\lambda$, and the true value can be estimated as the average over several such choices. We estimate the error of our computation from the standard error of the mean. The reader should perform such a computation as an exercise.

One can perform a fit of the points $|\Delta x_n|/\epsilon$ to the exponential function in the following way: Since $|\Delta x_n|/\epsilon \sim C\exp(\lambda n) \Rightarrow \ln(|\Delta x_n|/\epsilon) = \lambda n + c$, we can make a fit to a straight line instead. Using `gnuplot`, the relevant commands are:

```
gnuplot> fit [5:53] a*x+b \
         "<echo 500 3.6 0.2 1e-15 |./l;cat lia.dat"\
         using 1:(log($4)) via a,b
gnuplot> replot exp(a*x+b)
```

The command shown above fits the data to the function `a*x+b` by taking the 1st column and the logarithm of the 4th column (`using 1:(log($4))`) of the `stdout` of the command that we used for creating the previous plot. We choose data for which $5 \leq n \leq 53$ (`[5:53]`) and the fitting parameters are `a,b` (`via a,b`). The second line, adds the fitted function to the plot.

Now we are going to use equation (3.35) for calculating $\lambda$. This equation is approximately correct when (a) we have already reached the steady state and (b) in the large $n$ limit. For this reason we should study if we obtain a satisfactory convergence when we (a) "throw away" a number of `NTRANS` steps, (b) calculate the sum (3.35) for increasing `NSTEPS`$= n$ (c) calculate the sum (3.35) for many values of the initial point $x_0$. This has to be carefully repeated for all values of $r$ since each factor will contribute differently to the quality of the convergence: In regions that manifest chaotic behavior (large $\lambda$) convergence will be slower. The program can be found in the file `liapunov2.f90`:

```
!==========================================================
```

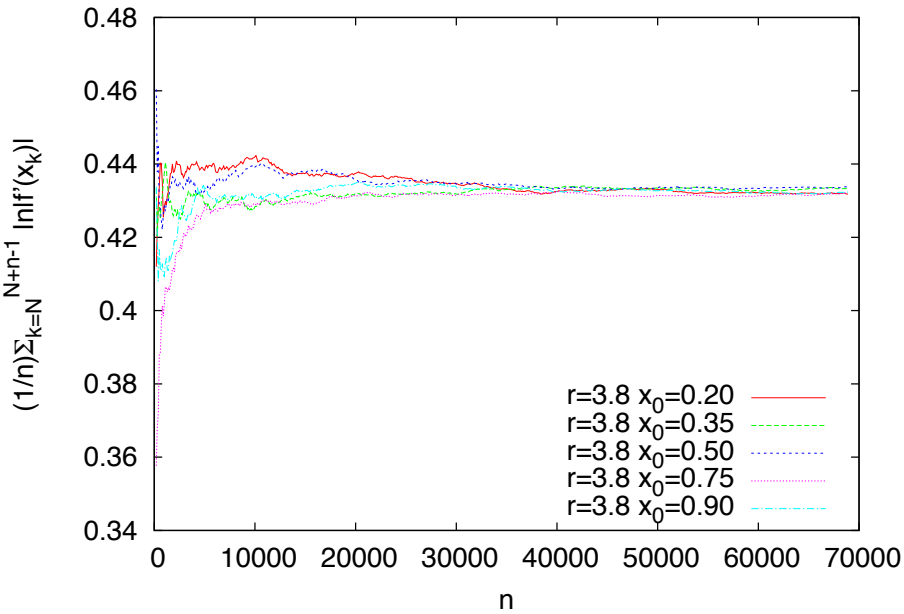



Figure 3.8: Plot of the sum $(1/n)\sum_{k=N}^{N+n-1} \ln|f'(x_k)|$ as a function of $n$ for the logistic map with $r = 3.8$, $N = 2000$ for different initial conditions $x_0 = 0.20, 0.35, 0.50, 0.75, 0.90$. The different curves converge in the limit $n \to \infty$ to $\lambda = 0.4325(10)$.

```
!Discrete Logistic Map:
!Liapunov exponent from sum_i ln|f'(x_i)|
! NTRANS: number of discarded iteration in order to discard
!         transient behavior
! NSTEPS: number of terms in the sum
!==========================================================
program logistic_map
 implicit none
 integer :: NTRANS,NSTEPS,i
 real(8) :: r,x0,x1,sum

! ------ Input:
 print *,'# Enter NTRANS,NSTEPS, r, x0:'
 read  *,NTRANS,NSTEPS,r,x0
 print *,'# NTRANS  = ',NTRANS
 print *,'# NSTEPS  = ',NSTEPS
 print *,'# r       = ',r
 print *,'# x0      = ',x0

 do i=1,NTRANS
  x1  = r * x0  * (1.0D0-x0 )
  x0  = x1
```

```
 enddo
 sum = log(ABS(r*(1.0D0-2.0D0*x0)))
! ------ Initialize:
 open(unit=33,file='lia.dat')
 write(33,*) 1,x0,sum
! ------ Calculate:
 do i=2,NSTEPS
  x1  = r * x0  * (1.0D0-x0 )
  sum = sum + log(ABS(r*(1.0D0-2.0D0*x1)))
  write(33,*)i,x1,sum/i
  x0  = x1
 enddo
 close(33)
end program logistic_map
```

After NTRANS steps, the program calculates NSTEPS times the sum of the terms $\ln|f'(x_k)| = \ln|r(1-2x_k)|$. At each step the sum divided by the number of steps i is printed to the file lia.dat. Figure 3.6 shows the results for $r = 3.8$. This is a point where the system exhibits strong chaotic behavior and convergence is achieved after we compute a large number of steps. Using NTRANS $= 2\,000$ and NSTEPS $\approx 70\,000$ the achieved accuracy is about 0.2% with $\lambda = 0.4325 \pm 0.0010 \equiv 0.4325(10)$. The main contribution to the error comes from the different paths followed by each initial point chosen. The plot can be constructed with the gnuplot commands:

```
> gfortran liapunov2.f90 -o l
> gnuplot
gnuplot> plot \
 "<echo 2000 70000 3.8 0.20 |./l;cat lia.dat" u 1:3 w l,\
 "<echo 2000 70000 3.8 0.35 |./l;cat lia.dat" u 1:3 w l,\
 "<echo 2000 70000 3.8 0.50 |./l;cat lia.dat" u 1:3 w l,\
 "<echo 2000 70000 3.8 0.75 |./l;cat lia.dat" u 1:3 w l,\
 "<echo 2000 70000 3.8 0.90 |./l;cat lia.dat" u 1:3 w l
```

The plot command runs the program using the parameters NTRANS $=$ $2\,000$, NSTEPS $= 70\,000$, r $= 3.8$ and x0 $= 0.20, 0.35, 0.50, 0.75, 0.90$ and plots the results from the contents of the file lia.dat.

In order to determine the regions of chaotic behavior we have to study the dependence of the Liapunov exponent $\lambda$ on the value of $r$. Using our experience coming from the careful computation of $\lambda$ before, we will run the program for several values of $r$ using the parameters NTRANS $= 2\,000$, NSTEPS $= 60\,000$ from the initial point x0 $= 0.2$. This calculation gives accuracy of the order of 1%. If we wish to measure $\lambda$ carefully and estimate the error of the results, we have to follow the steps described in

figures 3.7 and 3.8. The program can be found in the file `liapunov3.f90` and it is a simple modification of the previous program so that it can perform the calculation for many values of $r$.

```
!==========================================================
!Discrete Logistic Map:
!Liapunov exponent from sum_i ln|f'(x_i)|
!Calculation for r in [rmin,rmax] with RSTEPS steps
! RSTEPS: values or r studied: r=rmin+(rmax-rmin)/RSTEPS
! NTRANS: number of discarded iteration in order to discard
!         transient behavior
! NSTEPS: number of terms in the sum
! xstart: value of initial x0 for every r
!==========================================================
program logistic_map
 implicit none
 real(8),parameter :: rmin   = 2.5D0
 real(8),parameter :: rmax   = 4.0D0
 real(8),parameter :: xstart = 0.2D0
 integer,parameter :: RSTEPS = 1000
 integer,parameter :: NSTEPS = 60000
 integer,parameter :: NTRANS = 2000
 integer :: i,ir
 real(8) :: r,x0,x1,sum,dr

 open(unit=33,file='lia.dat')
 dr = (rmax-rmin)/(RSTEPS-1)
 do ir=0,RSTEPS-1
  r = rmin+ir*dr
  x0= xstart
  do i=1,NTRANS
   x1  = r * x0  * (1.0D0-x0 )
   x0  = x1
  enddo
  sum = log(ABS(r*(1.0D0-2.0D0*x0)))
! ----- Calculate:
  do i=2,NSTEPS
   x1  = r * x0  * (1.0D0-x0 )
   sum = sum + log(ABS(r*(1.0D0-2.0D0*x1)))
   x0  = x1
  enddo
  write(33,*)r,sum/NSTEPS
 enddo !do ir=0,RSTEPS-1
 close(33)
end program logistic_map
```

The program can be compiled and run using the commands:

```
> gfortran liapunov3.f90 -o l
> ./l &
```

The character & makes the program `./l` to run in the background. This is recommended for programs that run for a long time, so that the shell returns the prompt to the user and the program continues to run even after the shell is terminated.

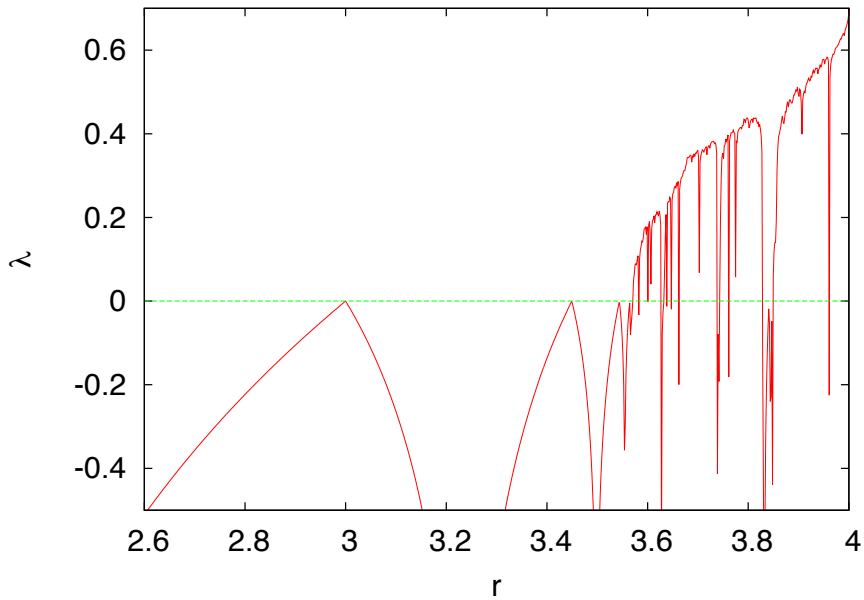



Figure 3.9: The Liapunov exponent $\lambda$ of the logistic map calculated via equation (3.35). Note the chaotic behavior that manifests for the values of $r$ where $\lambda > 0$ and the windows of stable $k$-cycles where $\lambda < 0$. Compare this plot with the bifurcation diagram of figure 3.4. At the points where $\lambda = 0$ we have onset of chaos (or "edge of chaos") with manifestation of weak chaos (i.e. $|\Delta x_n| \sim |\Delta x_0| n^{\omega}$). At these points we have transitions from stable $k$-cycles to strong chaos. We observe the onset of chaos for the first time when $r = r_c \approx 3.5699$, at which point $\lambda = 0$ (for smaller $r$ the plot seems to touch the $\lambda = 0$ line, but in fact $\lambda$ takes negative values with $|\lambda|$ very small).

The data are saved in the file `lia.dat` and we can make the plot shown in figure 3.7 using gnuplot:

```
gnuplot> plot "lia.dat" with lines notitle ,0 notitle
```

Now we can compare figure 3.9 with the bifurcation diagram shown in figure 3.4. The intervals with $\lambda < 0$ correspond to stable $k$-cycles. The

intervals where $\lambda > 0$ correspond to manifestation of *strong chaos*. These intervals are separated by points with $\lambda = 0$ where the system exhibits *weak chaos*. This means that neighboring trajectories diverge from each other with a power law $|\Delta x_n| \sim |\Delta x_0| n^\omega$ instead of an exponential, where $\omega = 1/(1-q)$ is a positive exponent that needs to be determined. The parameter $q$ is the one usually used in the literature. Strong chaos is obtained in the $q \to 1$ limit. For larger $r$, switching between chaotic and stable periodic trajectories is observed each time $\lambda$ changes sign. The critical values of $r$ can be computed with relatively high accuracy by restricting the calculation to a small enough neighborhood of the critical point. You can do this using the program listed above by setting the parameters `rmin` and `rmax`.

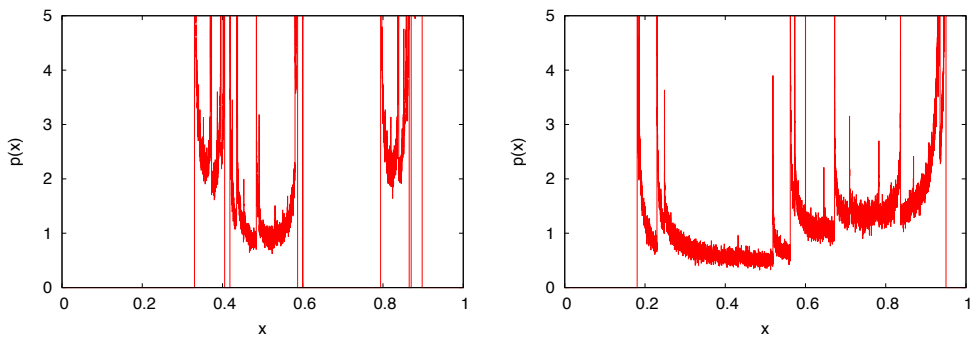



Figure 3.10: The distribution functions $p(x)$ of $x$ of the logistic map for $r = 3.59$ (left) and $3.8$ (right). The chaotic behavior appears to be weaker for $r = 3.59$, and this is reflected on the value of the entropy. One sees that there exist intervals of $x$ with $p(x) = 0$ which become smaller and vanish as $r$ gets close to 4. This distribution is very hard to be distinguished from a truly random distribution.

We can also study the chaotic properties of the trajectories of the logistic map by computing the distribution $p(x)$ of the values of $x$ in the interval $(0, 1)$. After the transitional period, the distribution $p(x)$ for the $k$ cycles will have support only at the points of the $k$ cycles, whereas for the chaotic regimes it will have support on subintervals of $(0, 1)$. The distribution function $p(x)$ is independent for most of the initial points of the trajectories. If one obtains a large number of points from many trajectories of the logistic map, it will be practically impossible to understand that these are produced by a deterministic rule. For this reason, chaotic systems can be used for the production of *pseudorandom* numbers, as we will see in chapter 11. By measuring the *entropy*, which is a measure of disorder in a system, we can quantify the "randomness" of

the distribution. As we will see in chapter 12, it is given by the equation

$$S = -\sum_k p_k \ln p_k \,, \tag{3.37}$$

where $p_k$ is the probability of observing the state $k$. In our case, we can make an approximate calculation of $S$ by dividing the interval $(0,1)$ to $N$ subintervals of width $\Delta x$. For given $r$ we obtain a large number $M$ of values $x_n$ of the logistic map and we compute the histogram $h_k$ of their distribution in the intervals $(x_k, x_k + \Delta x)$. The probability density is obtained from the limit of $p_k = h_k/(M\Delta x)$ as $M$ becomes large and $\Delta x$ small (large $N$). Indeed, $\sum_{k=1}^{N} p_k \Delta x = 1$ converges to $\int_0^1 p(x)\,dx = 1$. We will define $S = -\sum_{k=1}^{N} p_k \ln p_k \Delta x$.

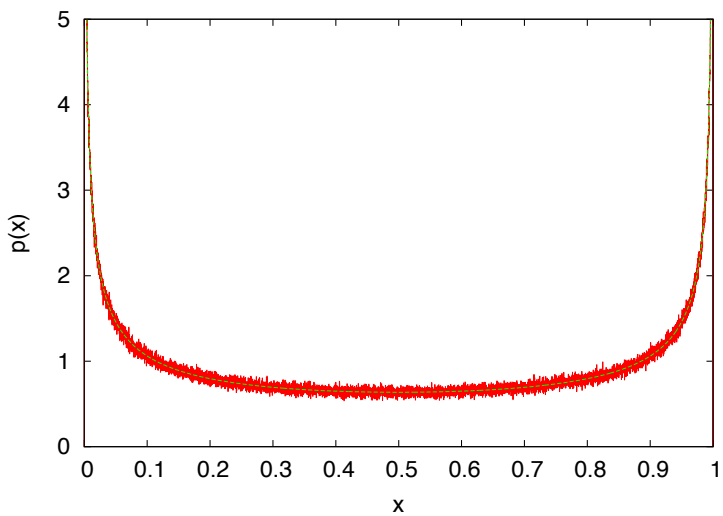



Figure 3.11: The distribution $p(x)$ of $x$ for the logistic map for $r = 4$. We observe strong chaotic behavior, $p(x)$ has support over the whole interval $(0,1)$ and the entropy is large. The solid line is the analytic form of the distribution $p(x) = \pi^{-1}x^{-1/2}(1-x)^{-1/2}$ which is known for $r = 4$ [27]. This is the beta distribution for $a = 1/2$, $b = 1/2$.

The program listed below calculates $p_k$ for chosen values of $r$, and then the entropy $S$ is calculated using (3.37). It is a simple modification of the program in `liapunov3.f90` where we add the parameter `NHIST` counting the number of intervals $N$ for the histograms. The probability density is calculated in the array `p(NHIST)`. The program can be found in the file `entropy.f90`:

```
!==========================================================
!Discrete Logistic Map:
!Entropy calculation from S=-sum_i p_i ln p_i
!Calculation for r in [rmin,rmax] with RSTEPS steps
! RSTEPS: values or r studied: r=rmin+(rmax-rmin)/RSTEPS
! NHIST : number of histogram bins for calculation of p_i
! NSTEPS: number of values of x in the histograms
! NTRANS: number of discarted iteration in order to discard
!         transient behavior
! xstart: value of initial x0 for every r
!==========================================================
program logistic_map
 implicit none
 real(8),parameter :: rmin   = 2.5D0
 real(8),parameter :: rmax   = 4.0D0
 real(8),parameter :: xstart = 0.2D0
 integer,parameter :: RSTEPS = 1000
 integer,parameter :: NHIST  = 10000
 integer,parameter :: NTRANS = 2000
 integer,parameter :: NSTEPS = 5000000
 real(8),parameter :: xmin=0.0D0,xmax=1.0D0
 integer :: i,ir,isum,n
 real(8) :: r,x0,x1,sum,dr,dx
 real(8) :: p(NHIST),S

 open(unit=33,file='entropy.dat')
 p  = 0.0D0
 dr = (rmax-rmin)/(RSTEPS-1)
 dx = (xmax-xmin)/(NHIST -1)
 do ir=0,RSTEPS-1
  r = rmin+ir*dr
  x0= xstart
  do i=1,NTRANS
   x1  = r * x0  * (1.0D0-x0 )
   x0  = x1
  enddo
!make histogram:
  n=INT(x0/dx)+1;p(n)=p(n)+1.0D0
  do i=2,NSTEPS
   x1  = r * x0  * (1.0D0-x0 )
   n   = INT(x1/dx)+1
   p(n)=p(n)+1.0D0
   x0  = x1
  enddo
!p(k) is now histogram of x-values.
!Normalize so that sum_k p(k)*dx=1
!to get probability distribution:
  p    = p/NSTEPS/dx
!sum all non zero terms: p(n)*log(p(n))*dx
```

```
  S     = -SUM(p*log(p),MASK=p.gt.0.0D0)*dx
  write(33,*)r,S
 enddo !do ir=0,RSTEPS-1
 close(33)
!print the last probability distribution:
 open(unit=34,file='entropy_hist.dat')
 do n=1,NHIST
  x0 = xmin +(n-1)*dx + 0.5D0*dx
  write(34,*) r,x0,p(n)
 enddo
 close(34)
end program logistic_map
```

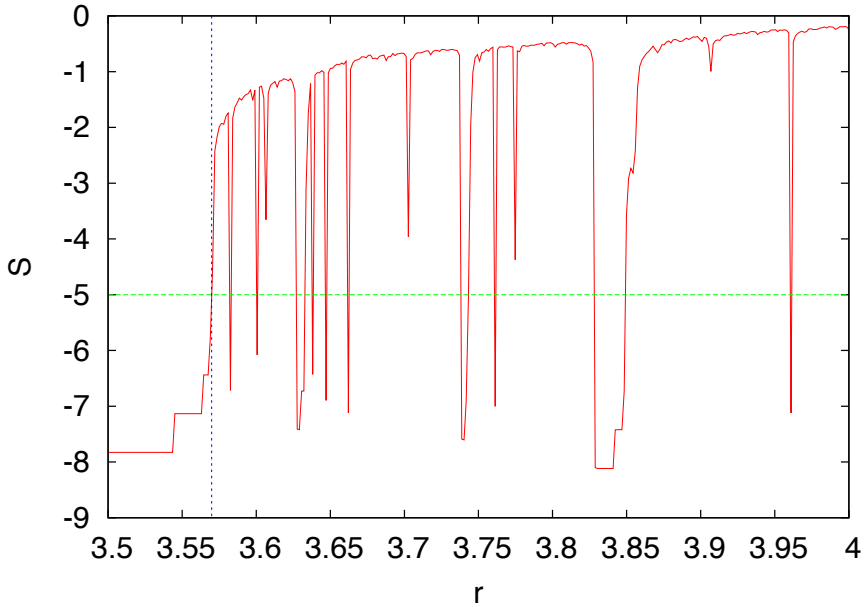



Figure 3.12: The entropy $S = -\sum_k p_k \ln p_k \, \Delta x$ for the logistic map as a function of $r$. The vertical line is $r_c \approx 3.56994567$ which marks the beginning of chaos and the horizontal is the corresponding entropy. The entropy is low for small values of $r$, where we have the stable $2^n$ cycles, and large in the chaotic regimes. $S$ drops suddenly when we pass to a (temporary) periodic behavior interval. We clearly observe the 3-cycle for $r = 1+\sqrt{8} \approx 3.8284$ and the subsequent bifurcations that we observed in the bifurcation diagram (figure 3.4) and the Liapunov exponent diagram (figure 3.9).

For the calculation of the distribution functions and the entropy we have to choose the parameters which control the systematic error. The parameter `NTRANS` should be large enough so that the transitional behavior will not contaminate our results. Our measurements must be checked

for being independent of its value. The same should be done for the initial point `xstart`. The parameter `NHIST` controls the partitioning of the interval $(0, 1)$ and the width $\Delta x$, so it should be large enough. The parameter `NSTEPS` is the number of "measurements" for each value of $r$ and it should be large enough in order to reduce the "noise" in $p_k$. It is obvious that `NSTEPS` should be larger when $\Delta x$ becomes smaller. Appropriate choices lead to the plots shown in figures 3.10 and 3.11 for $r = 3.59$, $3.58$ και $4$. We see that stronger chaotic behavior means a wider distribution of the values of $x$.

The entropy is shown in figure 3.12. The stable periodic trajectories lead to small entropy, whereas the chaotic ones lead to large entropy. There is a sudden increase in the value of the entropy at the beginning of chaos at $r = r_c$, which increases even further as the chaotic behavior becomes stronger. During the intermissions of the chaotic behavior there are sudden drops in the value of the entropy. It is quite instructive to compare the entropy diagrams with the corresponding bifurcation diagrams (see figure 3.4) and the Liapunov exponent diagrams (see figure 3.9). The entropy is increasing until $r$ reaches its maximum value 4, but this is not done smoothly. By magnifying the corresponding areas in the plot, we can see an infinite number of sudden drops in the entropy in intervals of $r$ that become more and more narrow.

## 3.7 Problems

Several of the programs that you need to write for solving the problems of this chapter can be found in the `Problems` directory of the accompanying software of this chapter.

3.1 Confirm that the trajectories of the logistic map for $r < 1$ are falling off exponentially for large enough $n$.

(a) Choose $r = 0.5$ and plot the trajectories for $x_0 = 0.1 - 0.9$ with step $0.1$ for $n = 1, \ldots, 1000$. Put the $y$ axis in a logarithmic scale. From the resulting curves discuss whether you obtain an exponential falloff.

(b) Fit the points $x_n$ for $n > 20$ to the function $c\,\mathrm{e}^{-ax}$ and determine the fitting parameters $a$ and $c$. How do these parameters depend on the initial point $x_0$? You can use the following `gnuplot` commands for your calculation:

```
gnuplot> !gfortran logistic.f90 -o l
gnuplot> a=0.7;c=0.4;
gnuplot> fit [10:] c*exp(-a*x) \
 "<echo 1000 0.5 0.5|./l;cat log.dat" via a,c
gnuplot> plot c*exp(-a*x),\
 "<echo 1000 0.5 0.5|./l;cat log.dat" w l
```

As you can see, we set `NSTEPS = 1000, r = 0.5, x0 = 0.5`. By setting the limits `[10:]` to the `fit` command, the fit includes only the points $x_n \geq 10$, therefore avoiding the transitional period and the deviation from the exponential falloff for small $n$.

(c) Repeat for $r = 0.3 - 0.9$ with step $0.1$ and for $r = 0.99, 0.999$. As you will be approaching $r = 1$, you will need to discard more points from near the origin. You might also need to increase `NSTEPS`. You should always check graphically whether the fitted exponential function is a good fit to the points $x_n$ for large $n$. Construct a table for the values of $a$ as a function of $r$.

The solutions of the equation (3.3) is $\mathrm{e}^{(r-1)x}$. How is this related to the values that you computed in your table?

3.2 Consider the logistic map for $r = 2$. Choose `NSTEPS=100` and calculate the corresponding trajectories for `x0=0.2, 0.3, 0.5, 0.7,`

0.9. Plot them on the same graph. Calculate the fixed point $x_2^*$ and compare your result to the known value $1 - 1/r$. Repeat for x0= $10^{-\alpha}$ for $\alpha = -1, -2, -5, -10, -20, -25$. What do you conclude about the point $x_1^* = 0$?

3.3 Consider the logistic map for $r = 2.9, 2.99, 2.999$. Calculate the stable point $x_2^*$ and compare your result to the known value $1 - 1/r$. How large should NSTEPS be chosen each time? You may choose x0=0.3.

3.4 Consider the logistic map for $r = 3.2$. Take x0=0.3, 0.5, 0.9 and NSTEPS=300 and plot the resulting trajectories. Calculate the fixed points $x_3^*$ and $x_4^*$ by using the command tail log.dat. Increase NSTEPS and repeat so that you make sure that the trajectory has converged to the 2-cycle. Compare their values to the ones given by equation (3.18). Make the following plots:

```
gnuplot> plot    \
 "<echo 300 3.2 0.3|./l;awk 'NR%2==0' log.dat"  w l
gnuplot> replot \
 "<echo 300 3.2 0.3|./l;awk 'NR%2==1' log.dat"  w l
```

What do you observe?

3.5 Repeat the previous problem for $r = 3.4494$. How big should NSTEPS be chosen so that you obtain $x_3^*$ and $x_4^*$ with an accuracy of 6 significant digits?

3.6 Repeat the previous problem for $r = 3.5$ and $3.55$. Choose NSTEPS = 1000, x0 = 0.5. Show that the trajectories approach a 4-cycle and an 8-cycle respectively. Calculate the fixed points $x_5^*$-$x_8^*$ and $x_9^*$-$x_{16}^*$.

3.7 Plot the functions $f(x)$, $f^{(2)}(x)$, $f^{(4)}(x)$, $x$ for given $r$ on the same graph. Use the commands:

```
gnuplot> set samples 1000
gnuplot> f(x) = r*x*(1-x)
gnuplot> r=1;plot [0:1] x,f(x),f(f(x)),f(f(f(f(x))))
```

The command r=1 sets the value of $r$. Take $r = 2.5, 3, 3.2, 1+\sqrt{6}, 3.5$. Determine the fixed points and the $k$-cycles from the intersections of the plots with the diagonal $y = x$.

3.8 Construct the cobweb plots of figures 3.2 and 3.4 for $r = 2.8, 3.3$ and $3.5$. Repeat by dropping from the plot an increasing number of initial points, so that in the end only the $k$-cycles will remain. Do the same for $r = 3.55$.

3.9 Construct the bifurcation diagrams shown in figure 3.4.

3.10 Construct the bifurcation diagram of the logistic map for $3.840 < r < 3.851$ and for $0.458 < x < 0.523$. Compute the first four bifurcation points with an accuracy of 5 significant digits by magnifying the appropriate parts of the plots. Take `NTRANS=15000`.

3.11 Construct the bifurcation diagram of the logistic map for $2.9 < r < 3.57$. Compute graphically the bifurcation points $r_c^{(n)}$ for $n = 2, 3, 4, 5, 6, 7, 8$. Make sure that your results are stable against variations of the parameters `NTRANS`, `NSTEPS` as well as from the choice of branching point. From the known values of $r_c^{(n)}$ for $n = 2, 3$, and from the dependence of your results on the choices of `NTRANS`, `NSTEPS`, estimate the accuracy achieved by this graphical method. Compute the ratios $(r_c^{(n)} - r_c^{(n-1)})/(r_c^{(n+1)} - r_c^{(n)})$ and compare your results to equation (3.20).

3.12 Choose the values of $\rho$ in equation (3.24) so that you obtain only one energy level. Compute the resulting value of the energy. When do we have three energy levels?

3.13 Consider the polynomial $g(x) = x^3 - 2x^2 - 11x + 12$. Find the roots obtained by the Newton-Raphson method when you choose $x_0 = 2.35287527, 2.35284172, 2.35283735, 2.352836327, 2.352836323$. What do you conclude concerning the basins of attraction of each root of the polynomial? Make a plot of the polynomial in a neighborhood of its roots and try other initial points that will converge to each one of the roots.

3.14 Use the Newton-Raphson method in order to compute the 4-cycle $x_5^*, \ldots, x_8^*$ of the logistic map. Use appropriate areas of the bifurcation diagram so that you can choose the initial points correctly. Check that your result for $r_c^{(4)}$ is the same for all $x_\alpha^*$. Tune the parameters chosen in your calculation on order to improve the accuracy of your measurements.

3.15 Repeat the previous problem for the 8-cycle $x_9^*, \ldots, x_{16}^*$ and $r_c^{(5)}$.

| $n$ | $r_c^{(n)}$ | $n$ | $r_c^{(n)}$ |
|---|---|---|---|
| 2 | 3.0000000000 | 10 | 3.56994317604 |
| 3 | 3.4494897429 | 11 | 3.569945137342 |
| 4 | 3.544090360 | 12 | 3.5699455573912 |
| 5 | 3.564407266 | 13 | 3.569945647353 |
| 6 | 3.5687594195 | 14 | 3.5699456666199 |
| 7 | 3.5696916098 | 15 | 3.5699456707464 |
| 8 | 3.56989125938 | 16 | 3.56994567163008 |
| 9 | 3.56993401837 | 17 | 3.5699456718193 |
| $r_c = 3.56994567\ldots$ | | | |

Table 3.1: The values of $r_c^{(n)}$ for the logistic map calculated for problem 17. $r_c^{(\infty)} \equiv r_c$ is taken from the bibliography.

3.16 Repeat the previous problem for the 16-cycle $x_{17}^*, \ldots, x_{32}^*$ and $r_c^{(6)}$.

3.17 Calculate the critical points $r_c^{(n)}$ for $n = 3, \ldots, 17$ of the logistic map using the Newton-Raphson method. In order to achieve that, you should determine the bifurcation points graphically in the bifurcation diagram first and then choose the initial points in the Newton-Raphson method appropriately. The program in `bifurcationPoints.f90` should read the parameters `eps, epsx, epsr` from the `stdin` so that they can be tuned for increasing $n$. If these parameters are too small the convergence will be unstable and if they are too large you will have large systematic errors. Using this method, try to reproduce table 3.1

3.18 Calculate the ratios $\Delta r^{(n)}/\Delta r^{(n+1)}$ of equation (3.20) using the results of table 3.1. Calculate Feigenbaum's constant and comment on the accuracy achieved by your calculation.

3.19 Estimate Feigenbaum's constant $\delta$ and the critical value $r_c$ by assuming that for large enough $n$, $r_c^{(n)} \approx r_c - C\delta^{-n}$. This behavior is a result of equation (3.20). Fit the results of table 3.1 to this function and calculate $\delta$ and $r_c$. This hypothesis is confirmed in figure 3.13 where we can observe the exponential convergence of $r_c^{(n)}$ to $r_c$. Construct the same plot using the parameters of your calculation.
Hint: You can use the following `gnuplot` commands:

```
gnuplot> nmin=2;nmax=17
gnuplot> r(x)= rc-c*d**(-x)
gnuplot> fit [nmin:nmax] r(x)  "rcrit" u 1:2 via rc,c,d
gnuplot> plot "rcrit", r(x)
gnuplot> print rc,d
```

The file `rcrit` contains the values of table 3.1. You should vary the parameters `nmin`, `nmax` and repeat until you obtain a stable fit.

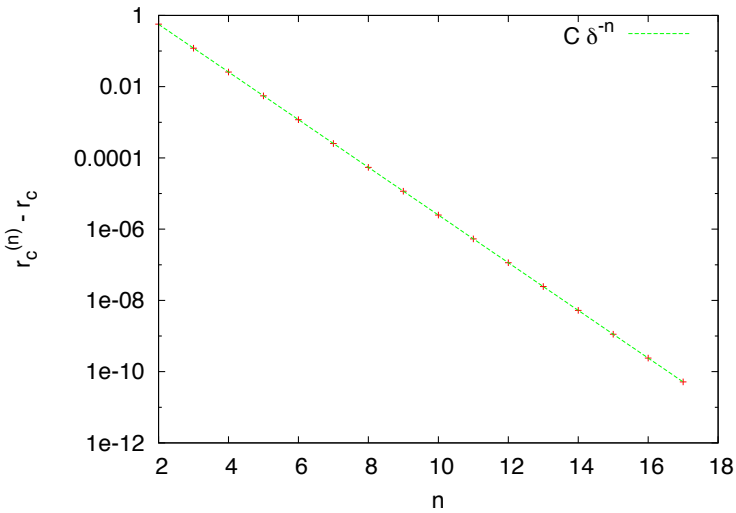



Figure 3.13: Test of the relation $r_c^{(n)} \approx r_c - C\delta^{-n}$ discussed in problem 17. The parameters used in the plot are approximately $r_c = 3.5699457$, $\delta = 4.669196$ and $C = 12.292$.

3.20 Use the Newton-Raphson method to calculate the first three bifurcation points after the appearance of the 3-cycle for $r = 1 + \sqrt{8}$. Choose one bifurcation point of the 3-cycle, one of the 6-cycle and one of the 12-cycle and magnify the bifurcation diagram in their neighborhood.

3.21 Consider the map describing the evolution of a population

$$x_{n+1} = p(x_n) = x_n \mathrm{e}^{r(1-x_n)} \,. \tag{3.38}$$

(a) Plot the functions $x$, $p(x)$, $p^{(2)}(x)$, $p^{(4)}(x)$ for $r = 1.8, 2, 2.6, 2.67, 2.689$ for $0 < x < 8$. For which values of $r$ do you expect to obtain stable $k$-cycles?

(b) For the same values of $r$ plot the trajectories with initial points $x_0 = 0.2, 0.5, 0.7$. For each $r$ make a separate plot.

(c) Use the Newton-Raphson method in order to determine the points $r_c^{(n)}$ for $n = 3, 4, 5$ as well as the first two bifurcation points of the 3-cycle.

(d) Construct the bifurcation diagram for $1.8 < r < 4$. Determine the point marking the onset of chaos as well as the point where the 3-cycle starts. Magnify the diagram around a branch that you will choose.

(e) Estimate Feigenbaum's constant $\delta$ as in problem 17. Is your result compatible with the expectation of universality for the value of $\delta$? Is the value of $r_c$ the same as that of the logistic map?

3.22 Consider the sine map:

$$x_{n+1} = s(x_n) = r \sin(\pi x_n) \,. \tag{3.39}$$

(a) Plot the functions $x$, $s(x)$, $s^{(2)}(x)$, $s^{(4)}(x)$, $s^{(8)}(x)$ for $r = 0.65$, $0.75$, $0.84$, $0.86$, $0.88$. Which values of $r$ are expected to lead to stable $k$-cycles?

(b) For the same values of $r$, plot the trajectories with initial points $x_0 = 0.2, 0.5, 0.7$. Make one plot for each $r$.

(c) Use the Newton-Raphson method in order to determine the points $r_c^{(n)}$ for $n = 3, 4, 5$ as well as the first two bifurcation points of the 3-cycle.

(d) Construct the bifurcation diagram for $0.6 < r < 1$. Within which limits do the values of $x$ lie in? Repeat for $0.6 < r < 2$. What do you observe? Determine the point marking the onset of chaos as well as the point where the 3-cycle starts. Magnify the diagram around a branch that you will choose.

3.23 Consider the map:

$$x_{n+1} = 1 - r x_n^2 \,. \tag{3.40}$$

(a) Construct the bifurcation diagram for $0 < r < 2$. Within which limits do the values of $x$ lie in? Determine the point marking the onset of chaos as well as the point where the 3-cycle starts. Magnify the diagram around a branch that you will choose.

(b) Use the Newton-Raphson method in order to determine the points $r_c^{(n)}$ for $n = 3, 4, 5$ as well as the first two bifurcation points of the 3-cycle.

3.24 Consider the tent map:

$$x_{n+1} = r \min\{x_n, 1 - x_n\} = \begin{cases} rx_n & 0 \le x_n \le \frac{1}{2} \\ r(1 - x_n) & \frac{1}{2} < x_n \le 1 \end{cases} . \tag{3.41}$$

Construct the bifurcation diagram for $0 < r < 2$. Within which limits do the values of $x$ lie in? On the same graph, plot the functions $r/2$, $r - r^2/2$.

Magnify the diagram in the area $1.407 < r < 1.416$ and $0.580 < x < 0.588$. At which point do the two disconnected intervals within which $x_n$ take their values merge into one? Magnify the areas $1.0 < r < 1.1$, $0.4998 < x < 0.5004$ and $1.00 < r < 1.03$, $0.4999998 < x < 0.5000003$ and determine the merging points of two disconnected intervals within which $x_n$ take their values.

3.25 Consider the Gauss map (or mouse map):

$$x_{n+1} = \mathrm{e}^{-rx_n^2} + q\,. \tag{3.42}$$

Construct the bifurcation diagram for $-1 < q < 1$ and $r = 4.5$, $4.9$, $7.5$. Make your program to take as the initial point of the new trajectory to be the last one of the previous trajectory and choose $x_0 = 0$ for $q = -1$. Repeat for $x_0 = 0.7, 0.5, -0.7$. What do you observe? Note that as $q$ is increased, we obtain bifurcations and "anti-bifurcations".

3.26 Consider the circle map:

$$x_{n+1} = [x_n + r - q\sin(2\pi x_n)] \mod 1\,. \tag{3.43}$$

(Make sure that your program keeps the values of $x_n$ so that $0 \le x_n < 1$). Construct the bifurcation diagram for $0 < q < 2$ and $r = 1/3$.

3.27 Use the program in `liapunov.f90` in order to compute the distance between two trajectories of the logistic map for $r = 3.6$ that originally are at a distance $\Delta x_0 = 10^{-15}$. Choose $x_0 = 0.1$, $0.2, 0.3$, $0.4$, $0.5, 0.6$, $0.7, 0.8$, $0.9$, $0.99, 0.999$ and calculate the Liapunov exponent by fitting to a straight line appropriately. Compute the mean value and the standard error of the mean.

3.28 Calculate the Liapunov exponent for $r = 3.58,\ 3.60,\ 3.65,\ 3.70,\ 3.80$ for the logistic map. Use both ways mentioned in the text. Choose at least 5 different initial points and calculate the mean and the standard error of the mean of your results. Compare the values of $\lambda$ that you obtain with each method and comment.

3.29 Compute the critical value $r_c$ numerically as the limit $\lim_{n\to\infty} r_c^{(n)}$ for the logistic map with an accuracy of nine significant digits. Use the calculation of the Liapunov exponent $\lambda$ given by equation (3.35).

3.30 Compute the values of $r$ of the logistic map numerically for which we (a) enter a stable 3-cycle (b) reenter into the chaotic behavior. Do the calculation by computing the Liapunov exponent $\lambda$ and compare your results with the ones obtained from the bifurcation diagram.

3.31 Calculate the Liapunov exponent using equation (3.35) for the following maps:

$$\begin{aligned}
x_{n+1} &= x_n \mathrm{e}^{r(1-x_n)}\,, \qquad 1.8 < r < 4 \\
x_{n+1} &= r\sin(\pi x_n)\,, \qquad 0.6 < r < 1 \\
x_{n+1} &= 1 - r x_n^2\,, \qquad 0 < r < 2 \\
x_{n+1} &= \mathrm{e}^{-r x_n^2} + q\,, \qquad r = 7.5, -1 < q < 1 \\
x_{n+1} &= \left[x_n + \frac{1}{3} - q\sin(2\pi x_n)\right] \mod 1\,, \quad 0 < q < 2\,, \qquad (3.44)
\end{aligned}$$

and construct the diagrams similar to the ones in figure 3.9. Compare your plots with the respective bifurcation diagrams (you may put both graphs on the same plot). Use two different initial points $x_0 = 0, 0.2$ for the Gauss map ($x_{n+1} = \mathrm{e}^{-r x_n^2} + q$) and observe the differences. For the circle map ($x_{n+1} = [x_n + 1/3 - q\sin(2\pi x_n)] \mod 1$) study carefully the values $0 < q < 0.15$.

3.32 Reproduce the plots in figures 3.10, 3.11 and 3.12. Compute the function $p(x)$ for $r = 3.68,\ 3.80,\ 3.93$ and $3.98$. Determine the points where you have stronger chaos by observing $p(x)$ and the corresponding values of the entropy. Compute the entropy for $r \in (3.95, 4.00)$ by taking `RSTEPS=2000` and estimate the values of $r$ where we enter to and exit from chaos. Compare your results with the computation of the Liapunov exponent.

3.33 Consider the Hénon map:

$$\begin{aligned} x_{n+1} &= y_n + 1 - a x_n^2 \\ y_{n+1} &= b x_n \end{aligned} \tag{3.45}$$

(a) Construct the two bifurcation diagrams for $x_n$ and $y_n$ for $b = 0.3$, $1.0 < a < 1.5$. Check if the values $a = 1.01, 1.4$ that we will use below correspond to stable periodic trajectories or chaotic behavior.

(b) Write a program in a file `attractor.f90` which will take `NINIT` = `NL` $\times$ `NL` initial conditions $(x_0(i), y_0(i))$ $i = 1, \ldots,$`NL` on a `NL`$\times$`NL` lattice of the square $x_m \leq x_0 \leq x_M$, $y_m \leq y_0 \leq y_M$. Each of the points $(x_0(i), y_0(i))$ will evolve according to equation (3.45) for $n =$ `NSTEPS` steps. The program will print the points $(x_n(i), y_n(i))$ to the `stdout`. Choose $x_m = y_m = 0.6$, $x_M = y_M = 0.8$, `NL`$= 200$.

(c) Choose $a = 1.01$, $b = 0.3$ and plot the points $(x_n(i), y_n(i))$ for $n = 0$, 1, 2, 3, 10, 20, 30, 40, 60, 1000 on the same diagram.

(d) Choose $a = 1.4$, $b = 0.3$ and plot the points $(x_n(i), y_n(i))$ for $n = 0, \ldots, 7$ on the same diagram.

(e) Choose $a = 1.4$, $b = 0.3$ and plot the points $(x_n(i), y_n(i))$ for $n = 999$ on the same diagram. Observe the Hénon strange attractor and its fractal properties. It is characterized by a Hausdorff[12] dimension $d_H = 1.261 \pm 0.003$. Then magnify the regions

$$\begin{array}{lll} \{(x,y)| & -1.290 < x < -1.270, & 0.378 < y < 0.384\}\,, \\ \{(x,y)| & 1.150 < x < -1.130, & 0.366 < y < 0.372\}\,, \\ \{(x,y)| & 0.108 < x < 0.114, & 0.238 < y < 0.241\}\,, \\ \{(x,y)| & 0.300 < x < 0.320, & 0.204 < y < 0.213\}\,, \\ \{(x,y)| & 1.076 < x < 1.084, & 0.090 < y < 0.096\}\,, \\ \{(x,y)| & 1.216 < x < 1.226, & 0.032 < y < 0.034\}\,. \end{array}$$

3.34 Consider the Duffing map:

$$\begin{aligned} x_{n+1} &= y_n \\ y_{n+1} &= -b x_n + a y_n - y_n^3\,. \end{aligned} \tag{3.46}$$

[12] D.A. Russel, J.D. Hanson, and E. Ott, "Dimension of strange attractors", Phys. Rev. Lett. **45** (1980) 1175. See "Hausdorff dimension" in Wikipedia.

(a) Construct the two bifurcation diagrams for $x_n$ and $y_n$ for $b = 0.3$, $0 < a < 2.78$. Choose four different initial conditions $(x_0, y_0) = (\pm 1/\sqrt{2}, \pm 1/\sqrt{2})$. What do you observe?

(b) Use the program `attractor.f90` from problem 33 in order to study the attractor of the map for $b = 0.3$, $a = 2.75$.

3.35 Consider the Tinkerbell map:

$$\begin{aligned} x_{n+1} &= x_n^2 - y_n^2 + a x_n + b y_n \\ y_{n+1} &= 2 x_n y_n + c x_n + d y_n \,. \end{aligned} \tag{3.47}$$

(a) Choose $a = 0.9$, $b = -0.6013$, $c = 2.0$, $d = 0.50$. Plot a trajectory on the plane by plotting the points $(x_n, y_n)$ for $n = 0, \ldots, 10\,000$ with $(x_0, y_0) = (-0.72, -0.64)$.

(b) Use the program `attractor.f90` from problem 33 in order to study the attractor of the map for the values of the parameters $a$, $b$, $c$, $d$ given above. Choose $x_m = -0.68$, $x_M = -0.76$, $y_m = -0.60$, $y_M = -0.68$, $n = 10\,000$.

(c) Repeat the previous question by taking $d = 0.27$.

# Chapter 4

# Motion of a Particle

In this chapter we will study the numerical solution of classical equations of motion of one dimensional mechanical systems, e.g. a point particle moving on the line, the simple pendulum etc. We will make an introduction to the numerical integration of ordinary differential equations with initial conditions and in particular to the Euler and Runge-Kutta methods. We study in detail the examples of the damped harmonic oscillator and of the damped pendulum under the influence of an external periodic force. The latter system is nonlinear and exhibits interesting chaotic behavior.

## 4.1 Numerical Integration of Newton's Equations

Consider the problem of the solution of the dynamical equations of motion of one particle under the influence of a dynamical field given by Newton's law. The equations can be written in the form

$$\frac{d^2\vec{x}}{dt^2} = \vec{a}(t, \vec{x}, \vec{v}) \,, \tag{4.1}$$

where

$$\vec{a}(t, \vec{x}, \vec{v}) \equiv \frac{\vec{F}}{m} \qquad \vec{v} = \frac{d\vec{x}}{dt} \,. \tag{4.2}$$

From the numerical analysis point of view, the problems that we will discuss are initial value problems for ordinary differential equations where the initial conditions

$$\vec{x}(t_0) = \vec{x}_0 \qquad \vec{v}(t_0) = \vec{v}_0 \,, \tag{4.3}$$

determine a unique solution $\vec{x}(t)$. The equations (4.1) are of second order with respect to time and it is convenient to write them as a system of twice as many first order equations:

$$\frac{d\vec{x}}{dt} = \vec{v} \qquad \frac{d\vec{v}}{dt} = \vec{a}(t, \vec{x}, \vec{v})\,. \tag{4.4}$$

In particular, we will be interested in the study of the motion of a particle moving on a line (1 dimension), therefore the above equations become

$$\begin{aligned} \frac{dx}{dt} &= v & \frac{dv}{dt} &= a(t, x, v) \quad \text{1-dimension} \\ x(t_0) &= x_0 & v(t_0) &= v_0\,. \end{aligned} \tag{4.5}$$

When the particle moves on the plane (2 dimensions) the equations of motion become

$$\begin{aligned} \frac{dx}{dt} &= v_x & \frac{dv_x}{dt} &= a_x(t, x, v_x, y, v_y) \quad \text{2-dimensions} \\ \frac{dy}{dt} &= v_y & \frac{dv_y}{dt} &= a_y(t, x, v_x, y, v_y) \\ x(t_0) &= x_0 & v_x(t_0) &= v_{0x} \\ y(t_0) &= y_0 & v_y(t_0) &= v_{0y}\,, \end{aligned} \tag{4.6}$$

## 4.2 Prelude: Euler Methods

As a first attempt to tackle the problem, we will study a simple pendulum of length $l$ in a homogeneous gravitational field $g$ (figure 4.1). The equations of motion are given by the differential equations

$$\begin{aligned} \frac{d^2\theta}{dt^2} &= -\frac{g}{l}\sin\theta \\ \frac{d\theta}{dt} &= \omega\,, \end{aligned} \tag{4.7}$$

which can be rewritten as a first order system of differential equations

$$\begin{aligned} \frac{d\theta}{dt} &= \omega \\ \frac{d\omega}{dt} &= -\frac{g}{l}\sin\theta \qquad , \end{aligned} \tag{4.8}$$

The equations above need to be written in a discrete form appropriate for a numerical solution with the aid of a computer. We split the interval

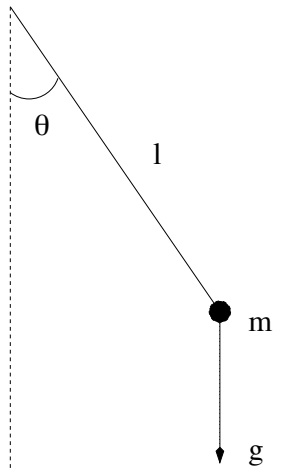



Figure 4.1: A simple pendulum of length $l$ in a homogeneous gravitational field $g$.

of time of integration $[t_i, t_f]$ to $N-1$ equal intervals[1] of width $\Delta t \equiv h$, where $h = (t_f - t_i)/(N-1)$. The derivatives are approximated by the relations $(x_{n+1} - x_n)/\Delta t \approx x'_n$, so that

$$\begin{aligned} \omega_{n+1} &= \omega_n + \alpha_n \Delta t \\ \theta_{n+1} &= \theta_n + \omega_n \Delta t \,. \end{aligned} \tag{4.9}$$

where $\alpha = -(g/l)\sin\theta$ is the angular acceleration. This is the so-called *Euler* method. The error at each step is estimated to be of order $(\Delta t)^2$. This is most easily seen by Taylor expanding around the point $t_n$ and neglecting all terms starting from the second derivative and beyond[2]. What we are mostly interested in is in the *total* error of the estimate of the functions we integrate for at time $t_f$! We expect that errors accumulate in an additive way at each integration step, and since the number of steps is $N \propto 1/\Delta t$ the total error should be $\propto (\Delta t)^2 \times (1/\Delta t) = \Delta t$. This is indeed what happens, and we say that Euler's method is a first order method. Its range of applicability is limited and we only study it for academic reasons. Euler's method is *asymmetric* because it uses information only from the beginning of the integration interval $(t, t+\Delta t)$. It can be put in a more balanced form by using the velocity *at the end* of the interval $(t, t+\Delta t)$. This way we obtain the Euler-Cromer method with a slightly

[1]We have $N$ discrete time points $t_i \equiv t_1, \ldots, t_{N-2}, t_{N-1} \equiv t_f$

[2]See appendix 4.7 for retails.

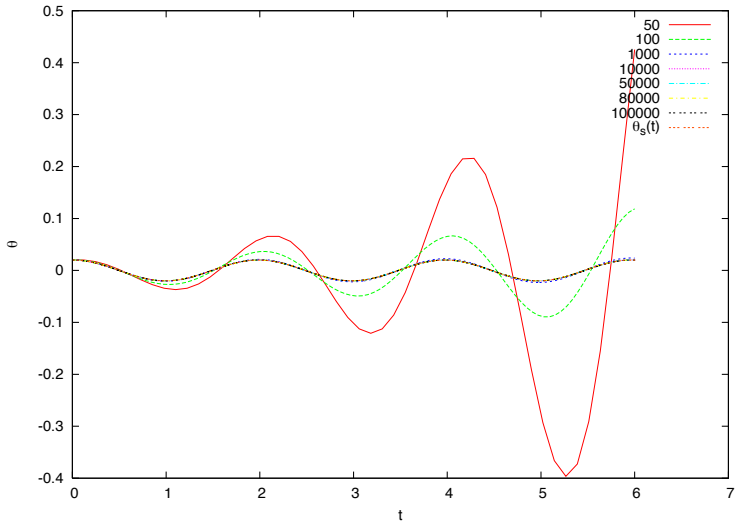



Figure 4.2: Convergence of Euler's method for a simple pendulum with period $T \approx 1.987 (\omega^2 = 10.0)$ for several values of the time step $\Delta t$ which is determined by the number of integration steps `Nt=` $50-100,000$. The solution is given for $\theta_0 = 0.2$, $\omega_0 = 0.0$ and we compare it with the known solution for small angles with $\alpha(t) \approx -(g/l)\,\theta$.

improved behavior, but which is still of first order

$$\begin{aligned} \omega_{n+1} &= \omega_n + \alpha_n \Delta t \\ \theta_{n+1} &= \theta_n + \omega_{n+1} \Delta t \,. \end{aligned} \tag{4.10}$$

An improved algorithm is the Euler–Verlet method which is of second order and gives total error[3] $\sim (\Delta t)^2$. This is given by the equations

$$\begin{aligned} \theta_{n+1} &= 2\theta_n - \theta_{n-1} + \alpha_n (\Delta t)^2 \\ \omega_n &= \frac{\theta_{n+1} - \theta_{n-1}}{2\Delta t} \,. \end{aligned} \tag{4.11}$$

The price that we have to pay is that we have to use a two step relation in order to advance the solution to the next step. This implies that we have to carefully determine the initial conditions of the problem which are given only at one given time $t_i$. We make one Euler time step *backwards* in order to define the value of $\theta_0$. If the initial conditions are $\theta_1 = \theta(t_i)$, $\omega_1 = \omega(t_i)$, then we *define*

$$\theta_0 = \theta_1 - \omega_1 \Delta t + \frac{1}{2} \alpha_1 (\Delta t)^2 \,. \tag{4.12}$$

[3]See appendix 4.7 for details.

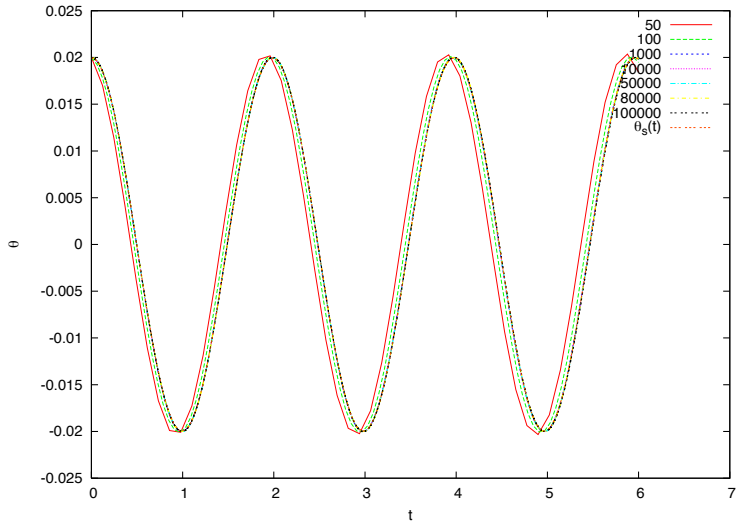



Figure 4.3: Convergence of the Euler-Cromer method, similarly to figure 4.2. We observe a faster convergence compared to Euler's method.

It is important that at this step the error introduced is not larger than $\mathcal{O}(\Delta t^2)$, otherwise it will spoil and eventually dominate the $\mathcal{O}(\Delta t^2)$ total error of the method introduced by the intermediate steps. At the last step we also have to take

$$\omega_N = \frac{\theta_N - \theta_{N-1}}{\Delta t} \,. \tag{4.13}$$

Even though the method has smaller total error than the Euler method, it becomes unstable for small enough $\Delta t$ due to roundoff errors. In particular, the second equation in (4.11) gives the angular velocity as the ratio of two small numbers. The problem is that the numerator is the result of the subtraction of two almost equal numbers. For small enough $\Delta t$, this difference has to be computed from the last digits of the finite representation of the numbers $\theta_{n+1}$ and $\theta_n$ in the computer memory. The accuracy in the determination of $(\theta_{n+1} - \theta_n)$ decreases until it eventually becomes *exactly* zero. For the first equation of (4.11), the term $\alpha_n \Delta t^2$ is smaller by a factor $\Delta t$ compared to the term $\alpha_n \Delta t$ in Euler's method. At some point, by decreasing $\Delta t$, we obtain $\alpha_n \Delta t^2 \ll 2\theta_n - \theta_{n-1}$ and the accuracy of the method vanishes due to the finite representation of real number in the memory of the computer. When the numbers $\alpha_n \Delta t^2$ and $2\theta_n - \theta_{n-1}$ differ from each other by more that approximately seven orders of magnitude, adding the first one to the second is equivalent to adding

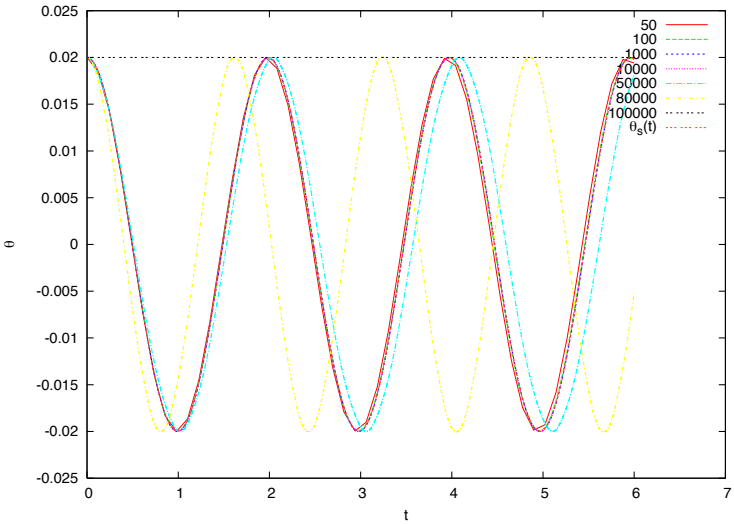



Figure 4.4: Convergence of the Euler-Verlet method, similarly to figure 4.2. We observe a faster convergence than Euler's method, but the roundoff errors make the results useless for `Nt`$\gtrsim 50,000$ (note what happens when `Nt`$= 100,000$. Why?).

zero and the contribution of the acceleration vanishes[4].

Writing programs that implement the methods discussed so far is quite simple. We will write a program that compares the results from all three methods Euler, Euler–Cromer and Euler–Verlet. The main program is mainly a user interface, and the computation is carried out by three subroutines `euler`, `euler_cromer` *και* `euler_verlet`. The user must provide the function `accel(x)` which gives the angular acceleration as a function of `x`. The variable `x` in our problem corresponds to the angle *theta*. For starters we take `accel(x)= -10.0 * sin(x)`, the acceleration of the simple pendulum.

The data structure is very simple: Three real arrays `REAL T(P), X(P)` and `V(P)` store the times $t_n$, the angles $\theta_n$ and the angular velocities $\omega_n$ *για* $n = 1, \ldots,$ `Nt`. The user determines the time interval for the integration from $t_i = 0$ to $t_f =$ `Tfi` and the number of discrete times `Nt`. The latter should be less than `P`, the size of the arrays. She also provides the initial conditions $\theta_0 =$ `Xin` and $\omega_0 =$ `Vin`. After this, we call the main integration

[4]Numbers of type `real` have approximately seven significant digits. The accuracy of the operations described above is determined by the number $\epsilon$, which is the smallest positive number such that $1 + \epsilon > 1$. For a variable `x` of some type, this number is given by a call to the Fortran intrinsic function `epsilon(x)`. For variables of type `real`, $\epsilon \approx 1.2 \times 10^{-7}$ and for variables of type `real(8)` $\epsilon \approx 2.2 \times 10^{-16}$.

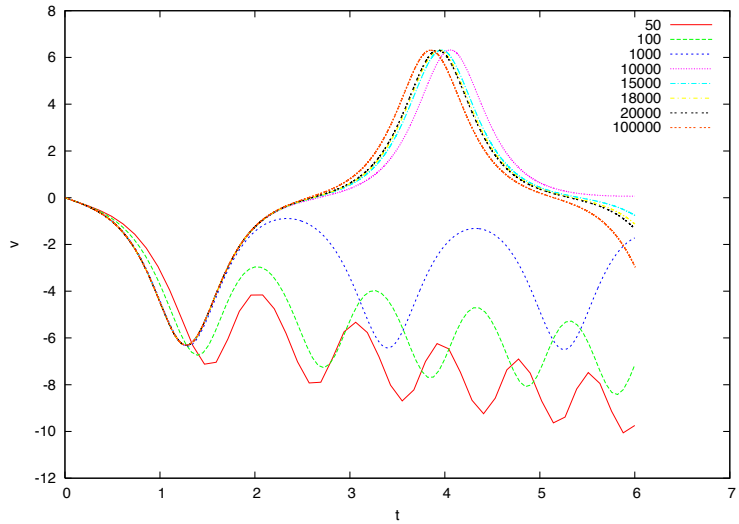



Figure 4.5: Convergence of Euler's method for the simple pendulum like in figure 4.2 for $\theta_0 = 3.0$, $\omega_0 = 0.0$. The behavior of the angular velocity is shown and we notice unstable behavior for $\texttt{Nt} \lesssim 1,000$.

routines which take as input the initial conditions, the time interval of the integration and the number of discrete times `Xin,Vin,Tfi,Nt`. The output of the routines is the arrays `T,X,V` which store the results for the time, position and velocity respectively. The results are printed to the files `euler.dat`, `euler_cromer.dat` and `euler_verlet.dat`.

After setting the initial conditions and computing the time step $\Delta t \equiv \texttt{h} = \texttt{Tfi}/(\texttt{Nt} - 1)$, the integration in each of the subroutines is performed in do loops which advance the solution for time $\Delta t$. The results are stored at each step in the arrays `T,X,V`. For example, the central part of the program for Euler's method is:

```
T(1) = 0.0
X(1) = Xin
V(1) = Vin
h    = Tfi/(Nt-1)
do i = 2,Nt
 T(i) = T(i-1)+h
 X(i) = X(i-1)+V(i-1)*h
 V(i) = V(i-1)+accel(X(i-1))*h
enddo
```

Some care has to be taken in the case of the Euler–Verlet method where one has to initialize the first two steps, as well as take special care for the

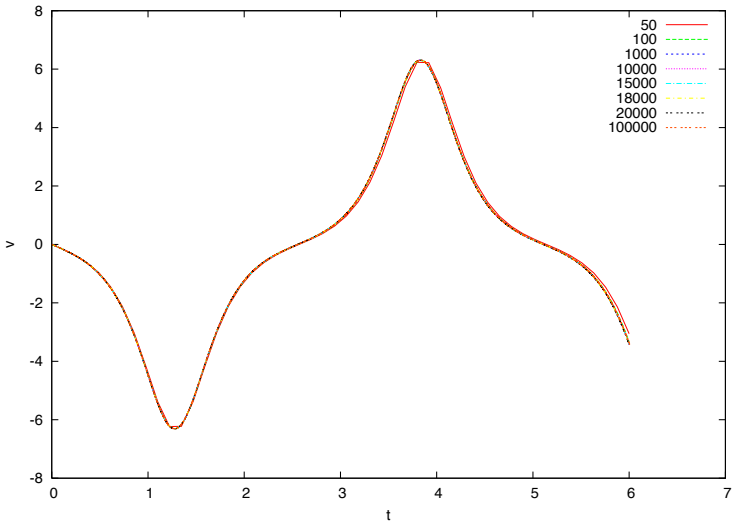



Figure 4.6: Convergence of Euler-Cromer's method, like in figure 4.5. We observe a faster convergence than for Euler's method.

last step for the velocity:

```
T(1)     = 0.0
X(1)     = Xin
V(1)     = Vin
X0       =     X(1)    - V(1) * h + accel(X(1))  *h*h/2.0
T(2)     = h
X(2)     = 2.0*X(1)    - X0       + accel(X(1))  *h*h
do i     = 3,Nt
  ..............
enddo
V(Nt)= (X(Nt)-X(Nt-1))/h
```

The full program can be found in the file `euler.f90` and is listed below:

```
!=========================================================
!Program to integrate equations of motion for accelerations
!which are functions of x with the method of Euler,
!Euler-Cromer and Euler-Verlet.
!The user sets initial conditions and the subroutines return
!X(t) and V(t)=dX(t)/dt in arrays T(1..Nt),X(1..Nt),V(1..Nt)
!The user provides number of times Nt and the final
!time Tfi. Initial time is assumed to be t_i=0 and the
!integration step h = Tfi/(Nt-1)
!The user programs a real function accel(x) which gives the
```

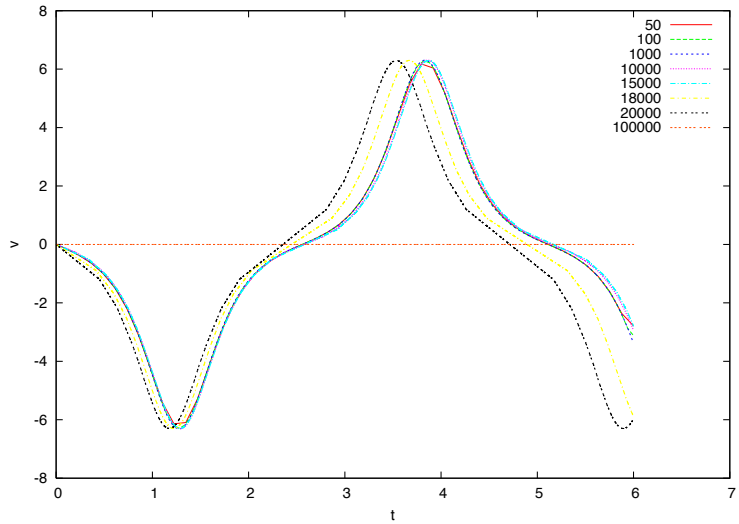



Figure 4.7: Convergence of the Euler-Verlet method, similarly to figure 4.5. We observe a faster convergence compared to Euler's method but that the roundoff errors make the results unstable for Nt$\gtrsim 18,000$.

```
!acceleration  dV(t)/dt as function of X.
!NOTE: T(1) = 0 T(Nt) = Tfi
!==========================================================
program diff_eq_euler
 implicit none
 integer,parameter:: P=110000 ! The size of the arrays
 real,dimension(P):: T,X,V    ! time t,x(t),v(t)=dx/dt
 real      :: Xin,Vin,Tfi     ! initial conditions
 integer   :: Nt,i
!The user provides initial conditions X_0,V_0 final time t_f
!and Nt:
 print *,'Enter X_0,V_0,t_f,Nt (t_i=0):'
 read(5,*)Xin,Vin,Tfi,Nt
!This check is necessary in order to avoid memory
!access violations:
 if(Nt .ge. P )then
  print *,'Nt must be strictly less than P. Nt,P= ',Nt,P
  stop
 endif
!Xin= X(1), Vin=V(1), T(1)=0 and the routine gives evolution in
!T(2..Nt), X(2..Nt), V(2..Nt) which we print in a file
 call euler(Xin,Vin,Tfi,Nt,T,X,V)
 open(unit=20,file="euler.dat")
 do i=1,Nt
!Each line in data file has time, position, velocity:
```

```
  write(20,*) T(i),X(i),V(i)
 enddo
 close(20) !we close the unit to be reused below
!-----------------------------------
!We repeat everything for each method
 call euler_cromer(Xin,Vin,Tfi,Nt,T,X,V)
 open(unit=20,file="euler_cromer.dat")
 do i=1,Nt
  write(20,*) T(i),X(i),V(i)
 enddo
 close(20)
!-----------------------------------
 call euler_verlet(Xin,Vin,Tfi,Nt,T,X,V)
 open(unit=20,file="euler_verlet.dat")
 do i=1,Nt
  write(20,*) T(i),X(i),V(i)
 enddo
 close(20)
!-----------------------------------
end program diff_eq_euler
!=========================================================
!Function which returns the value of acceleration at
!position x used in the integration subroutines
!euler, euler_cromer and euler_verlet
!=========================================================
real function accel(x)
 implicit none
 real x
 accel = -10.0*sin(x)
end function accel
!=========================================================
!Driver routine for integrating equations of motion
!using the Euler method
!Input:
!Xin=X(1), Vin=V(1) -- initial condition at t=0,
!Tfi the final time and Nt the number of times
!Output:
!The arrays T(1..Nt), X(1..Nt), V(1..Nt) which
!gives x(t_k)=X(k), dx/dt(t_k)=V(k), t_k=T(k) k=1..Nt
!where for k=1 we have the initial condition.
!=========================================================
subroutine euler(Xin,Vin,Tfi,Nt,T,X,V)
 implicit none
 integer :: Nt
 real,dimension(Nt) :: T,X,V    !time t,x(t),v(t)=dx/dt
 real    :: Xin,Vin,Tfi
 integer :: i
 real    :: h,accel     !**declare the function accel**
!Initial conditions set here:
```

```
 T(1) = 0.0
 X(1) = Xin
 V(1) = Vin
!h is the time step Dt
 h    = Tfi/(Nt-1)
 do i = 2,Nt
  T(i) = T(i-1)+h           ! time advances by Dt=h
  X(i) = X(i-1)+V(i-1)*h    ! advancement of position
  V(i) = V(i-1)+accel(X(i-1))*h !and velocity.
 enddo

end subroutine euler
!=========================================================
!Driver routine for integrating equations of motion
!using the Euler-Cromer method
!Input:
!Xin=X(1), Vin=V(1) -- initial condition at t=0,
!Tfi the final time and Nt the number of times
!Output:
!The arrays T(1..Nt), X(1..Nt), V(1..Nt) which
!gives x(t_i)=X(i), dx/dt(t_i)=V(i), t_i=T(i) i=1..Nt
!where for i=1 we have the initial condition.
!=========================================================
subroutine euler_cromer(Xin,Vin,Tfi,Nt,T,X,V)
 implicit none
 integer :: Nt
 real,dimension(Nt):: T,X,V !time t,x(t),v(t)=dx/dt
 real    :: Xin,Vin,Tfi
 integer :: i
 real    :: h,accel

 T(1) = 0.0
 X(1) = Xin
 V(1) = Vin
 h    = Tfi/(Nt-1)
 do i = 2,Nt
  T(i) = T(i-1)+h
  V(i) = V(i-1)+accel(X(i-1))*h
  !here is the difference compared to Euler
  X(i) = X(i-1)+V(i)*h
 enddo

end subroutine euler_cromer
!=========================================================
!Driver routine for integrating equations of motion
!using the Euler - Verlet method
!Input:
!Xin=X(1), Vin=V(1) -- initial condition at t=0,
!Tfi the final time and Nt the number of times
```

```
!Output:
!The arrays T(1..Nt), X(1..Nt), V(1..Nt) which
!gives x(t_i)=X(i), dx/dt(t_i)=V(i), t_i=T(i) i=1..Nt
!where for i=1 we have the initial condition.
!==========================================================
subroutine euler_verlet(Xin,Vin,Tfi,Nt,T,X,V)
 implicit none
 integer :: Nt
 real,dimension(Nt):: T,X,V !time t,x(t),v(t)=dx/dt
 real    :: Xin,Vin,Tfi
 integer :: i
 real    :: h,h2,X0,o2h
 real    :: accel
!Initial conditions set here:
 T(1)    = 0.0
 X(1)    = Xin
 V(1)    = Vin
 h       = Tfi/(Nt-1)   ! time step
 h2      = h*h             ! time step squared
 o2h     = 0.5/h           ! h/2
!We have to initialize one more step: X0 corresponds to 'X(0)'
 X0      =     X(1)   - V(1) * h + accel(X(1))  *h2/2.0
 T(2)    = h
 X(2)    = 2.0*X(1)   - X0       + accel(X(1))  *h2
!Now i starts from 3:
 do i    = 3,Nt
  T(i)   = T(i-1)+h
  X(i)   = 2.0*X(i-1) - X(i-2)   + accel(X(i-1))*h2
  V(i-1) = o2h * (X(i)-X(i-2))
 enddo
!Notice that we have one more step for the velocity:
 V(Nt)= (X(Nt)-X(Nt-1))/h
end subroutine euler_verlet
```

Compiling the running the program can be done with the commands:

```
> gfortran euler.f90 -o euler
> ./euler
 Enter X_0,V_0,t_f,Nt (t_i=0):
0.2 0.0 6.0 1000
> ls  euler*.dat
euler_cromer.dat  euler.dat  euler_verlet.dat
> head -n 5 euler.dat
   0.000000       0.2000000        0.000000
  6.0060062E-03  0.2000000      -1.1932093E-02
  1.2012012E-02  0.1999283      -2.3864185E-02
  1.8018018E-02  0.1997850      -3.5792060E-02
  2.4024025E-02  0.1995700      -4.7711499E-02
```

The last command shows the first 5 lines of the file `euler.dat`. We see the data for the time, the position and the velocity stored in 3 columns. We can graph the results using `gnuplot`:

```
gnuplot> plot "euler.dat" using 1:2 with lines
gnuplot> plot "euler.dat" using 1:3 with lines
```

These commands result in plotting the positions and the velocities as a function of time respectively. We can add the results of all methods to the last plot with the commands:

```
gnuplot> replot "euler_cromer.dat" using 1:3 with lines
gnuplot> replot "euler_verlet.dat" using 1:3 with lines
```

The results can be seen in figures 4.2–4.7. Euler's method is unstable unless we take a quite small time step. The Euler–Cromer method behaves impressively better. The results converge and remain constant for `Nt`$\sim 100,000$. The Euler–Verlet method converges much faster, but roundoff errors kick in soon. This is more obvious in figure 4.7 where the initial angular position is large. For small angles we can compare with the solution one obtains for the harmonic pendulum ($\sin(\theta) \approx \theta$):

$$\begin{aligned} \alpha(\theta) &= -\frac{g}{l}\,\theta \equiv -\Omega^2\,\theta \\ \theta(t) &= \theta_0 \cos(\Omega t) + (\omega_0/\Omega)\sin(\Omega t) \\ \omega(t) &= \omega_0 \cos(\Omega t) - (\theta_0\Omega)\sin(\Omega t)\,. \end{aligned} \tag{4.14}$$

In figures 4.2–4.4 we observe that the results agree with the above formulas for the values of $\Delta t$ where the methods converge. This way we can check our program for bugs. The plot of the functions above can be done with the following `gnuplot` commands[5]:

```
gnuplot> set dummy t
gnuplot> omega2  = 10
gnuplot> X0      = 0.2
gnuplot> V0      = 0.0
gnuplot> omega   = sqrt(omega2)
gnuplot> x(t)    = X0 * cos(omega * t) +(V0/omega)*sin(omega*t)
gnuplot> v(t)    = V0 * cos(omega * t) -(omega*X0)*sin(omega*t)
gnuplot> plot x(t), v(t)
```

[5]The command `set dummy t` sets the independent variable for functions to be `t` instead of `x` which is the default.

The results should not be compared only graphically since subtle differences can remain unnoticed. It is more desirable to plot the *differences* of the theoretical values from the numerically computed ones which can be done using the commands:

```
gnuplot> plot "euler.dat" using 1:($2-x($1)) with lines
gnuplot> plot "euler.dat" using 1:($3-v($1)) with lines
```

The command `using 1:($2-x($1))` puts the values found in the first column on the $x$ axis and the value found in the second column minus the value of the function `x(t)` for $t$ equal to the value found in the first column on the $y$ axis. This way, we can make the plots shown in[6] figures 4.11-4.14.

## 4.3 Runge–Kutta Methods

Euler's method is a one step finite difference method of first order. First order means that the total error introduced by the discretization of the integration interval $[t_i, t_f]$ by $N$ discrete times is of order $\sim \mathcal{O}(h)$, where $h \equiv \Delta t = (t_f - t_i)/N$ is the time step of the integration. In this section we will discuss a generalization of this approach where the total error will be of higher order in $h$. This is the class of Runge-Kutta methods which are one step algorithms where the total discretization error is of order $\sim \mathcal{O}(h^p)$. The local error introduced at each step is of order $\sim \mathcal{O}(h^{p+1})$ leading after $N = (t_f - t_i)/\Delta t$ steps to a maximum error of order

$$\sim \mathcal{O}(h^{p+1}) \times N = \mathcal{O}(h^{p+1}) \times \frac{t_f - t_i}{\Delta t} \sim \mathcal{O}(h^{p+1}) \times \frac{1}{h} = \mathcal{O}(h^p)\,. \tag{4.15}$$

In such a case we say that we have a Runge-Kutta method of $p^{th}$ order. The price one has to pay for the increased accuracy is the evaluation of the derivatives of the functions in more than one points in the interval $(t, t + \Delta t)$.

Let's consider for simplicity the problem with only one unknown function $x(t)$ which evolves in time according to the differential equation:

$$\frac{dx}{dt} = f(t, x)\,. \tag{4.16}$$

Consider the first order method first. The most naive approach would

[6] A small modification is necessary in order to plot the *absolute* value of the differences.

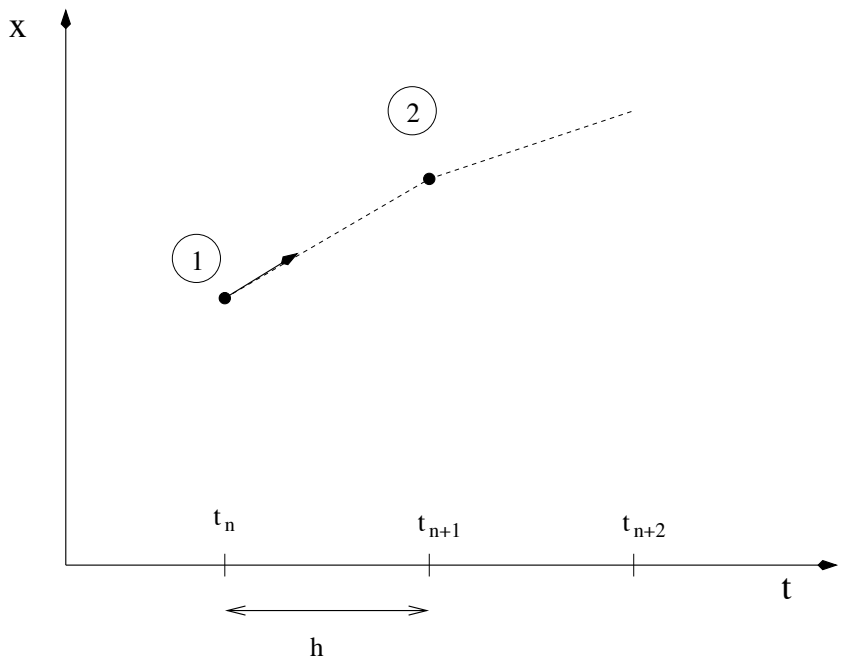



Figure 4.8: The geometry of the step of the Runge-Kutta method of $1^{st}$ order given by equation (4.17).

be to take the derivative to be given by the finite difference

$$\frac{dx}{dt} \approx \frac{x_{n+1} - x_n}{\Delta t} = f(t_n, x_n) \Rightarrow x_{n+1} = x_n + hf(t_n, x_n)\,. \tag{4.17}$$

By Taylor expanding, we see that the error at each step is $\mathcal{O}(h^2)$, therefore the error after integrating from $t_i \to t_f$ is $\mathcal{O}(h)$. Indeed,

$$x_{n+1} = x(t_n + h) = x_n + h\frac{dx}{dt}(x_n) + \mathcal{O}(h^2) = x_n + hf(t_n, x_n) + \mathcal{O}(h^2)\,. \tag{4.18}$$

The geometry of the step is shown in figure 4.8. We start from point 1 and by linearly extrapolating in the direction of the derivative $k_1 \equiv f(t_n, x_n)$ we determine the point $x_{n+1}$.

We can improve the method above by introducing an intermediate point 2. This process is depicted in figure 4.9. We take the point 2 in the middle of the interval $(t_n, t_{n+1})$ by making a linear extrapolation from $x_n$ in the direction of the derivative $k_1 \equiv f(t_n, x_n)$. Then we use the slope at point 2 as an *estimator* of the derivative within this interval, i.e. $k_2 \equiv f(t_{n+1/2}, x_{n+1/2}) = f(t_n + h/2, x_n + (h/2)k_1)$. We use $k_2$ to linearly

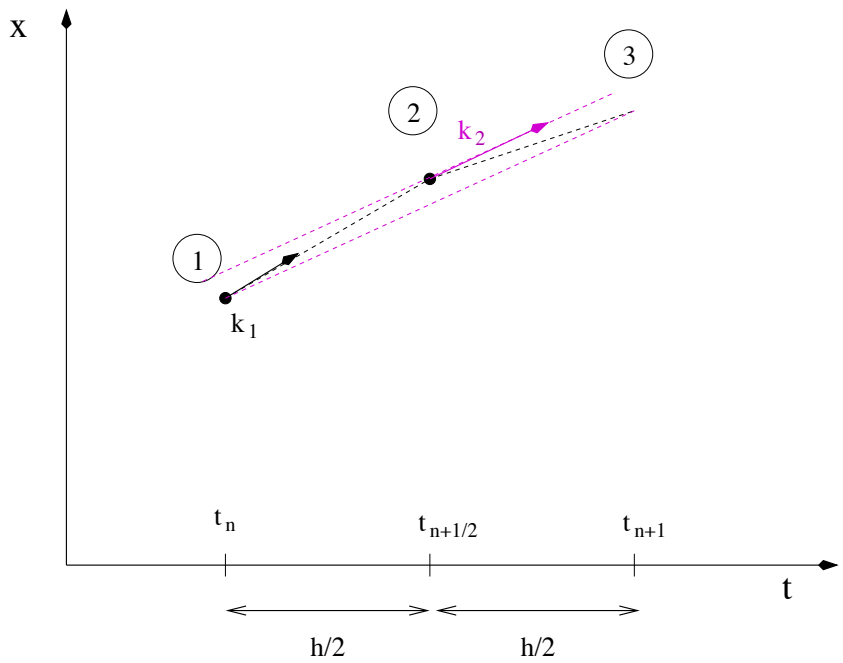



Figure 4.9: The geometry of an integration step of the 2nd order Runge-Kutta method given by equation (4.19).

extrapolate from $x_n$ to $x_{n+1}$. Summarizing, we have that

$$\begin{aligned} k_1 &= f(t_n, x_n) \\ k_2 &\equiv f(t_n + \frac{h}{2}, x_n + \frac{h}{2}\, k_1) \\ x_{n+1} &= x_n + h k_2 \,. \end{aligned} \tag{4.19}$$

For the procedure described above we have to evaluate $f$ twice at each step, thereby doubling the computational effort. The error at each step (4.19) becomes $\sim \mathcal{O}(h^3)$, however, giving a total error of $\sim \mathcal{O}(h^2) \sim \mathcal{O}(1/N^2)$. So for given computational time, (4.19) is superior to (4.17).

We can further improve the accuracy gain by using the Runge–Kutta method of 4th order. In this case we have 4 evaluations of the derivative $f$ per step, but the total error becomes now $\sim \mathcal{O}(h^4)$ and the method is superior to that of (4.19)[7]. The process followed is explained geometrically in figure 4.10. We use 3 intermediate points for evolving the solution from $x_n$ to $x_{n+1}$. Point 2 is determined by linearly extrapolating from $x_n$

[7]Not always though! Higher order does not necessarily mean higher accuracy, although this is true in the simple cases considered here.

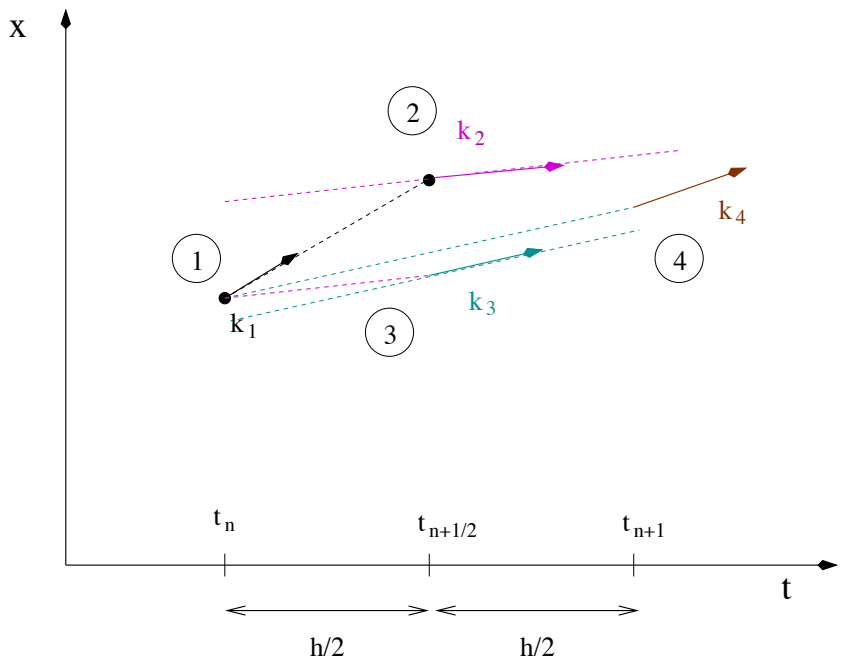



Figure 4.10: The geometry of an integration step of the Runge-Kutta method of 4th order given by equation (4.20).

to the midpoint of the interval $(t_n, t_{n+1} = t_n + h)$ by using the direction given by the derivative $k_1 \equiv f(t_n, x_n)$, i.e. $x_2 = x_n + (h/2)k_1$. We calculate the derivative $k_2 \equiv f(t_n + h/2, x_n + (h/2)k_1)$ at the point 2 and we use it in order to determine point 3, also located at the midpoint of the interval $(t_n, t_{n+1})$. Then we calculate the derivative $k_3 \equiv f(t_n + h/2, x_n + (h/2)k_2)$ at the point 3 and we use it to linearly extrapolate to the *end* of the interval $(t_n, t_{n+1})$, thereby obtaining point 4, i.e. $x_4 = x_n + hk_3$. Then we calculate the derivative $k_4 \equiv f(t_n + h, x_n + hk_3)$ at the point 4, and we use all four derivative $k_1, k_2, k_3$ and $k_4$ as estimators of the derivative of the function in the interval $(t_n, t_{n+1})$. If each derivative contributes with a particular weight in this estimate, the discretization error can become

$\sim \mathcal{O}(h^5)$. Such a choice is

$$
\begin{aligned}
k_1 &= f(t_n, x_n) \\
k_2 &= f(t_n + \frac{h}{2}, x_n + \frac{h}{2}\, k_1) \\
k_3 &= f(t_n + \frac{h}{2}, x_n + \frac{h}{2}\, k_2) \\
k_4 &= f(t_n + h, x_n + h\, k_3) \\
x_{n+1} &= x_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)\,. \qquad (4.20)
\end{aligned}
$$

We note that the second term of the last equation takes an average of the four derivatives with weights $1/6$, $1/3$, $1/3$ and $1/6$ respectively. A generic small change in these values will increase the discretization error to worse than $h^5$.

We remind to the reader the fact that by decreasing $h$ the discretization errors decrease, but that roundoff errors will start showing up for small enough $h$. Therefore, a careful determination of $h$ that minimizes the total error should be made by studying the dependence of the results as a function of $h$.

### 4.3.1 A Program for the 4th Order Runge–Kutta

Consider the problem of the motion of a particle in one dimension. For this, we have to integrate a system of two differential equations (4.5) for two unknown functions of time $x_1(t) \equiv x(t)$ and $x_2(t) \equiv v(t)$ so that

$$
\frac{dx_1}{dt} = f_1(t, x_1, x_2) \qquad \frac{dx_2}{dt} = f_2(t, x_1, x_2) \qquad (4.21)
$$

In this case, equations (4.20) generalize to:

$$
\begin{aligned}
k_{11} &= f_1(t_n, x_{1,n}, x_{2,n}) \\
k_{21} &= f_2(t_n, x_{1,n}, x_{2,n}) \\
k_{12} &= f_1(t_n + \frac{h}{2}, x_{1,n} + \frac{h}{2}\,k_{11}, x_{2,n} + \frac{h}{2}\,k_{21}) \\
k_{22} &= f_2(t_n + \frac{h}{2}, x_{1,n} + \frac{h}{2}\,k_{11}, x_{2,n} + \frac{h}{2}\,k_{21}) \\
k_{13} &= f_1(t_n + \frac{h}{2}, x_{1,n} + \frac{h}{2}\,k_{12}, x_{2,n} + \frac{h}{2}\,k_{22}) \\
k_{23} &= f_2(t_n + \frac{h}{2}, x_{1,n} + \frac{h}{2}\,k_{12}, x_{2,n} + \frac{h}{2}\,k_{22}) \\
k_{14} &= f_1(t_n + h, x_{1,n} + h\,k_{13}, x_{2,n} + h\,k_{23}) \\
k_{24} &= f_2(t_n + h, x_{1,n} + h\,k_{13}, x_{1,n} + h\,k_{23}) \\
x_{1,n+1} &= x_{1,n} + \frac{h}{6}(k_{11} + 2k_{12} + 2k_{13} + k_{14}) \\
x_{2,n+1} &= x_{1,n} + \frac{h}{6}(k_{21} + 2k_{22} + 2k_{23} + k_{24})\,. \qquad (4.22)
\end{aligned}
$$

Programming this algorithm is quite simple. The main program is an interface between the user and the driver routine of the integration. The user enters the initial and final times $t_i =$ `Ti` and $t_f =$ `Tf` and the number of discrete time points `Nt`. The initial conditions are $x_1(t_i) =$ `X10`, $x_2(t_i) =$ `X20`. The main data structure consists of three `real` arrays `T(P)`, `X1(P)`, `X2(P)` which store the times $t_i \equiv t_1, t_2, \ldots, t_{\texttt{Nt}} \equiv t_f$ and the corresponding values of the functions $x_1(t_k)$ and $x_2(t_k)$, $k = 1, \ldots,$ `Nt`. The main program calls the driver routine `RK(T,X1,X2,Ti,Tf,X10,X20,Nt)` which "drives" the heart of the program, the subroutine `RKSTEP(t,x1,x2,dt)` which performs one integration step using equations (4.22). `RKSTEP` evolves the functions `x1`, `x2` at time `t` by one step $h =$ `dt`. The routine `RK` stores the calculated values in the arrays `T`, `X1` and `X2` at each step. When `RK` returns the control to the main program, all the results are stored in `T`, `X1` and `X2`, which are subsequently printed in the file `rk.dat`. The full program is listed below and can be found in the file `rk.f90`:

```
!=========================================================
!Program to solve a 2 ODE system using Runge-Kutta Method
!User must supply derivatives
!dx1/dt=f1(t,x1,x2) dx2/dt=f2(t,x1,x2)
!as real functions
!Output is written in file rk.dat
!=========================================================
```

```
program rk_solve
 implicit none
 integer, parameter :: P=110000
 real,dimension(P)  :: T,X1,X2
 real     :: Ti,Tf,X10,X20
 integer :: Nt
 integer :: i
!Input:
 print *,'Runge-Kutta Method for 2-ODEs Integration'
 print *,'Enter Nt,Ti,TF,X10,X20:'
 read  *, Nt,Ti,Tf,X10,X20
 print *,'Nt = ',Nt
 print *,'Time: Initial Ti =',Ti,' Final Tf=',Tf
 print *,'            X1(Ti)=',X10,' X2(Ti)=',X20
 if(Nt.gt.P) stop 'Nt>P'
!The Calculation:
 call RK(T,X1,X2,Ti,Tf,X10,X20,Nt)
!Output:
 open(unit=11,file='rk.dat')
 do i=1,Nt
  write(11,*)T(i),X1(i),X2(i)
 enddo
 close(11)

end program rk_solve
!=========================================================
!The functions f1,f2(t,x1,x2) provided by the user
!=========================================================
real function f1(t,x1,x2)
 implicit none
 real :: t,x1,x2
 f1=x2           !dx1/dt= v = x2
end function f1
!---------------------------------------------------------
real function f2(t,x1,x2)
 implicit none
 real :: t,x1,x2
 f2=-10.0D0*x1   !harmonic oscillator
end function f2
!=========================================================
!RK(T,X1,X2,Ti,Tf,X10,X20,Nt) is the driver
!for the Runge-Kutta integration routine RKSTEP
!Input: Initial and final times Ti,Tf
!       Initial values at t=Ti  X10,X20
!       Number of steps of integration: Nt-1
!       Size of arrays T,X1,X2
!Output: real arrays T(Nt),X1(Nt),X2(Nt) where
!T(1) = Ti X1(1) = X10 X2(1) = X20
!          X1(k) = X1(at t=T(k)) X2(k) = X2(at t=T(k))
```

```
!T(Nt)=TF
!=========================================================
subroutine RK(T,X1,X2,Ti,Tf,X10,X20,Nt)
 implicit none
 integer :: Nt
 real,dimension(Nt):: T,X1,X2
 real     :: Ti,Tf,X10,X20
 real     :: dt
 real     :: TS,X1S,X2S !values of time and X1,X2 at given step
 integer :: i
!Initialize variables:
 dt       = (Tf-Ti)/(Nt-1)
 T (1)    = Ti
 X1(1)    = X10
 X2(1)    = X20
 TS       = Ti
 X1S      = X10
 X2S      = X20
!Make RK steps: The arguments of RKSTEP
!are replaced with the new ones!
 do i=2,Nt
  call RKSTEP(TS,X1S,X2S,dt)
  T (i)   = TS
  X1(i)   = X1S
  X2(i)   = X2S
 enddo
end subroutine RK
!=========================================================
!Subroutine RKSTEP(t,x1,x2,dt)
!Runge-Kutta Integration routine of ODE
!dx1/dt=f1(t,x1,x2) dx2/dt=f2(t,x1,x2)
!User must supply derivative functions:
!real function f1(t,x1,x2)
!real function f2(t,x1,x2)
!Given initial point (t,x1,x2) the routine advances it
!by time dt.
!Input : Inital time t    and function values x1,x2
!Output: Final  time t+dt and function values x1,x2
!Careful!: values of t,x1,x2 are overwritten...
!=========================================================
subroutine RKSTEP(t,x1,x2,dt)
 implicit none
 real :: t,x1,x2,dt
 real :: f1,f2
 real :: k11,k12,k13,k14,k21,k22,k23,k24
 real :: h,h2,h6

 h  =dt       !h =dt, integration step
 h2 =0.5D0*h !h2=h/2
```

```
 h6 =h/6.0    !h6=h/6

 k11=f1(t,x1,x2)
 k21=f2(t,x1,x2)
 k12=f1(t+h2,x1+h2*k11,x2+h2*k21)
 k22=f2(t+h2,x1+h2*k11,x2+h2*k21)
 k13=f1(t+h2,x1+h2*k12,x2+h2*k22)
 k23=f2(t+h2,x1+h2*k12,x2+h2*k22)
 k14=f1(t+h  ,x1+h  *k13,x2+h  *k23)
 k24=f2(t+h  ,x1+h  *k13,x2+h  *k23)

 t  =t+h
 x1 =x1+h6*(k11+2.0D0*(k12+k13)+k14)
 x2 =x2+h6*(k21+2.0D0*(k22+k23)+k24)

end subroutine RKSTEP
```

## 4.4 Comparison of the Methods

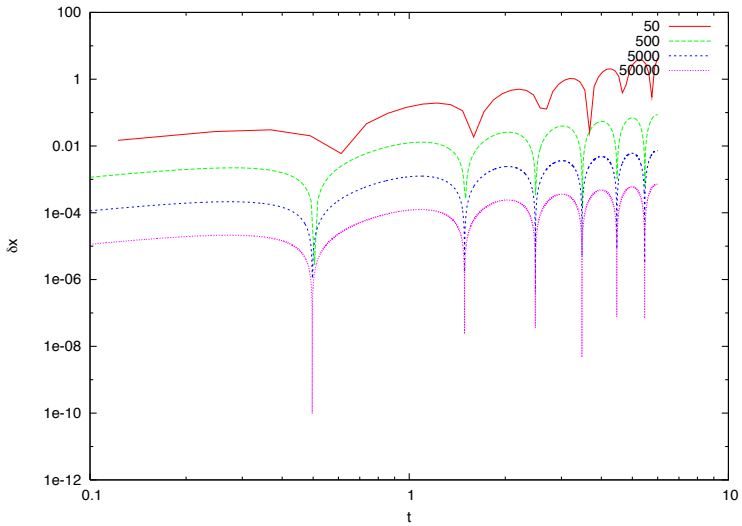



Figure 4.11: The discrepancy of the numerical results of the Euler method from the analytic solution for the simple harmonic oscillator. The parameters chosen are $\omega^2 = 10$, $t_i = 0$, $t_f = 6$, $x(0) = 0.2$, $v(0) = 0$ and the number of steps is $N = 50, 500, 5,000, 50,000$. Observe that the error becomes approximately ten times smaller each time according to the expectation of being of order $\sim \mathcal{O}(\Delta t)$.

In this section we will check our programs for correctness and accuracy w.r.t. discretization and roundoff errors. The simplest test is to

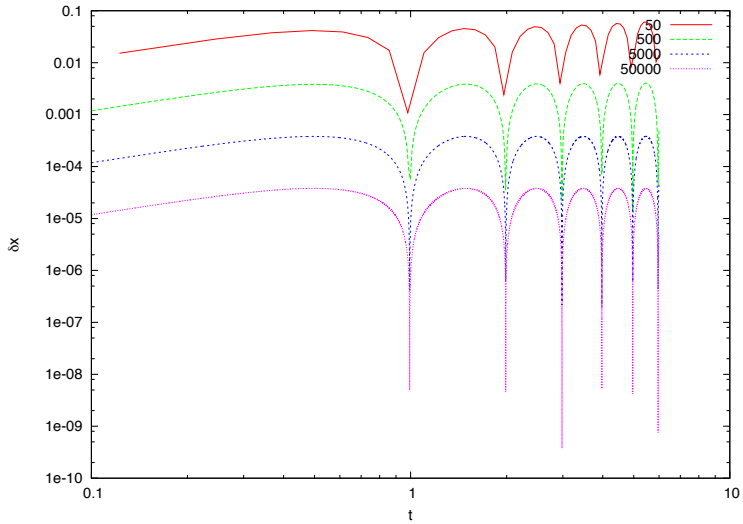



Figure 4.12: Like in figure 4.11 for the Euler-Cromer method. The error becomes approximately ten times smaller each time according to the expectation of being of order $\sim \mathcal{O}(\Delta t)$.

check the results against a known analytic solution of a simple model. This will be done for the simple harmonic oscillator. Our programs will need small changes which are summarized below. First, we will need to use higher accuracy variables and we will change all variables of type `REAL` to `REAL(8)`. For this we need to change the corresponding declarations in the beginning of each (sub)program. For each numerical constant in the program we need to put an explicit exponent with the letter `D` instead of an `E`. For example `0.5` → `0.5D0`, `1.2E-3` → `1.2D-3` etc. Then we need to alter the functions that compute the acceleration of the particle to give $a = -\omega^2 x$. We will take $\omega^2 = 10$ ($T \approx 1.987$). Therefore the relevant part of the program in `euler.f90` becomes

```
real(8) function accel(x)
 implicit none
 real(8) :: x
 accel = -10.0D0*x
end function accel
```

and that of the program in `rk.f90` becomes

```
real(8) function f2(t,x1,x2)
 implicit none
```

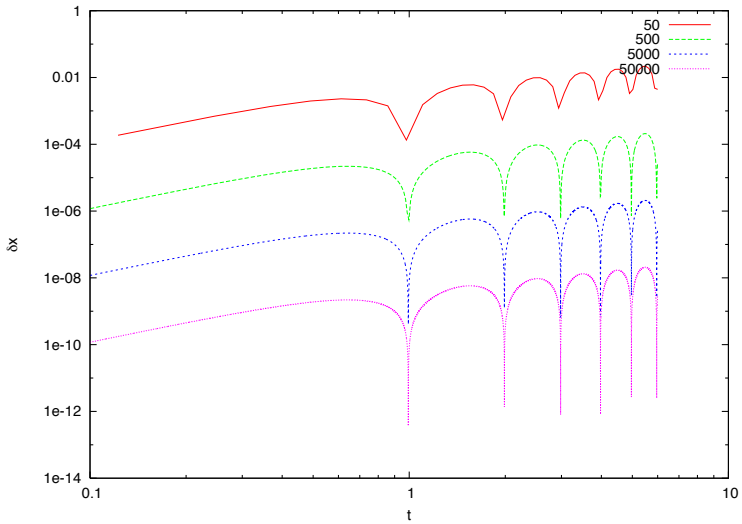



Figure 4.13: Like in figure 4.11 for the Euler-Verlet method. The error becomes approximately 100 times smaller each time according to the expectation of being of order $\sim \mathcal{O}(\Delta t^2)$.

```
 real(8) :: t,x1,x2
 f2=-10.0D0*x1
end function f2
```

The programs are run for a given time interval $t_i = 0$ to $t_f = 6$ with the initial conditions $x_0 = 0.2$, $v_0 = 0$. The time step $\Delta t$ is varied by varying the number of steps `Nt-1`. The computed numerical solution is compared to the well known solution for the simple harmonic oscillator

$$\begin{aligned} a(x) &= -\omega^2 x \\ x_h(t) &= x_0 \cos(\omega t) + (v_0/\omega) \sin(\omega t) \\ v_h(t) &= v_0 \cos(\omega t) - (x_0 \omega) \sin(\omega t)\,, \end{aligned} \tag{4.23}$$

We study the deviation $\delta x(t) = |x(t) - x_h(t)|$ and $\delta v(t) = |v(t) - v_h(t)|$ as a function of the time step $\Delta t$. The results are shown in figures 4.11–4.14. We note that for the Euler method and the Euler–Cromer method, the errors are of order $\mathcal{O}(\Delta t)$ as expected. However, the latter has smaller errors compared to the first one. For the Euler–Verlet method, the error turns out to be of order $\mathcal{O}(\Delta t^2)$ whereas for the 4th order Runge–Kutta is of order[8] $\mathcal{O}(\Delta t^4)$.

[8]The reader should confirm these claims, initially by looking at the figures 4.11-4.14 and then by reproducing these results. A particular time $t$ can be chosen and the errors can be plotted against $\Delta t$, $\Delta t^2$ and $\Delta t^4$ respectively.

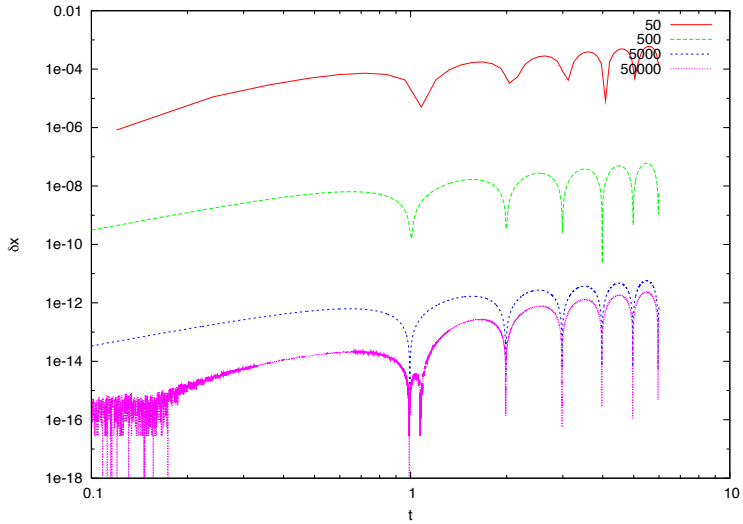



Figure 4.14: Like in figure 4.11 for the 4th order Runge–Kutta method. The error becomes approximately $10^{-4}$ times smaller each time according to the expectation of being of order $\sim \mathcal{O}(\Delta t^4)$. The roundoff errors become apparent for $50,000$ steps.

Another way for checking the numerical results is by looking at a conserved quantity, like the energy, momentum or angular momentum, and study its deviation from its original value. In our case we study the mechanical energy

$$E = \frac{1}{2}mv^2 + \frac{1}{2}m\omega^2 x^2 \tag{4.24}$$

which is computed at each step. The deviation $\delta E = |E - E_0|$ is shown in figures 4.15–4.18.

## 4.5 The Forced Damped Oscillator

In this section we will study a simple harmonic oscillator subject to a damping force proportional to its velocity and an external periodic driving force, which for simplicity will be taken to have a sinusoidal dependence in time,

$$\frac{d^2x}{dt^2} + \gamma\frac{dx}{dt} + \omega_0^2 x = a_0 \sin\omega t\,, \tag{4.25}$$

where $F(t) = ma_0 \sin\omega t$ and $\omega$ is the angular frequency of the driving force.

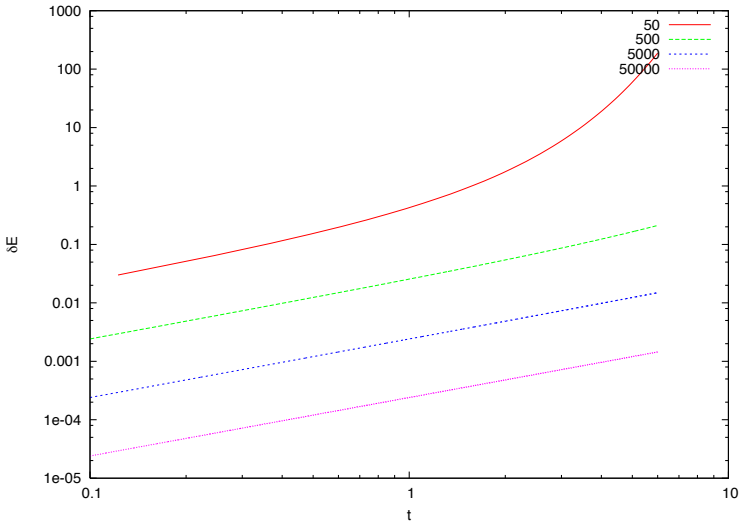



Figure 4.15: Like in figure 4.11 for the case of mechanical energy for the Euler method.

Consider initially the system without the influence of the driving force, i.e. with $a_0 = 0$. The real solutions of the differential equation[9] which are finite for $t \to +\infty$ are given by

$$x_0(t) = c_1 \mathrm{e}^{-(\gamma+\sqrt{\gamma^2-4\omega_0^2})t/2} + c_2 \mathrm{e}^{-(\gamma-\sqrt{\gamma^2-4\omega_0^2})t/2}\,, \quad \gamma^2 - 4\omega_0^2 > 0\,, \tag{4.26}$$

$$x_0(t) = c_1 \mathrm{e}^{-\gamma t/2} + c_2 \mathrm{e}^{-\gamma t/2} t\,, \quad \gamma^2 - 4\omega_0^2 = 0\,, \tag{4.27}$$

$$\begin{aligned} x_0(t) &= c_1 \mathrm{e}^{-\gamma t/2} \cos\left(\sqrt{-\gamma^2 + 4\omega_0^2}\; t/2\right) \\ &\quad + c_2 \mathrm{e}^{-\gamma t/2} \sin\left(\sqrt{-\gamma^2 + 4\omega_0^2}\; t/2\right)\,, \quad \gamma^2 - 4\omega_0^2 < 0 \end{aligned} \tag{4.28}$$

In the last case, the solution oscillates with an amplitude decreasing exponentially with time.

In the $a_0 > 0$ case, the general solution is obtained from the sum of a special solution $x_s(t)$ and the solution of the homogeneous equation $x_0(t)$. A special solution can be obtained from the ansatz $x_s(t) = A \sin \omega t +$

[9]These are easily obtained by substituting the ansatz $x(t) = A\mathrm{e}^{-\Omega t}$ and solving for $\Omega$.

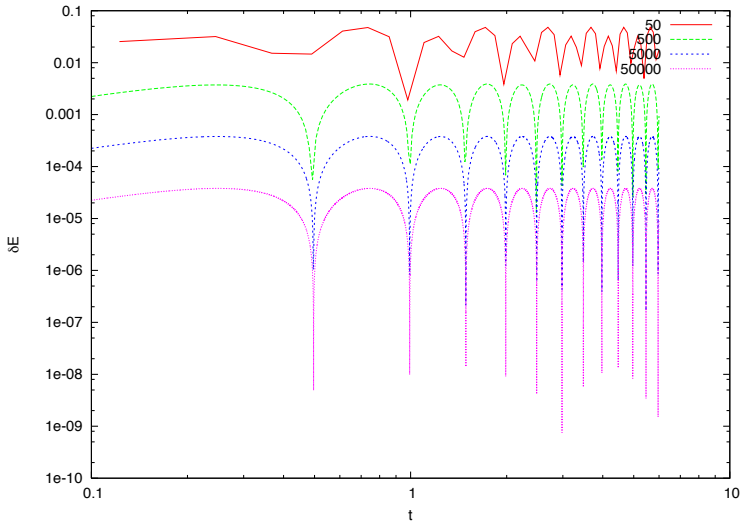



Figure 4.16: Like in figure 4.11 for the case of mechanical energy for the Euler–Cromer method.

$B\cos\omega t$, which when substituted in (4.25) and solved for $A$ and $B$ we find that

$$x_s(t) = \frac{a_0\left[(\omega_0^2-\omega^2)\cos\omega t + \gamma\omega\sin\omega t\right]}{(\omega_0^2-\omega^2)^2+\omega^2\gamma^2}\,, \tag{4.29}$$

and

$$x(t) = x_0(t) + x_s(t)\,. \tag{4.30}$$

The solution $x_0(t)$ decreases exponentially with time and eventually only $x_s(t)$ remains. The only case where this is not true, is when we have resonance without damping for $\omega=\omega_0$, $\gamma=0$. In that case the solution is

$$x(t) = c_1\cos\omega t + c_2\sin\omega t + \frac{a_0}{4\omega^2}\left(\cos\omega t + 2(\omega t)\sin\omega t\right)\,. \tag{4.31}$$

The first two terms are the same as that of the simple harmonic oscillator. The last one increases the amplitude linearly with time, which is a result of the influx of energy from the external force to the oscillator.

Our program will be a simple modification of the program in `rk.f90`. The main routines `RK(T,X1,X2,T0,TF,X10,X20,Nt)` and `RKSTEP(t,x1,x2,dt)` remain as they are. We only change the user interface. The basic parameters $\omega_0$, $\omega$, $\gamma$, $a_0$ are entered interactively by the user from the standard input `stdin`. These parameters should be accessible also by the function `f2(t,x1,x2)`, and one way to be able to do this, is to store them in variables which are placed in a *common block*. Such variables are accessible to

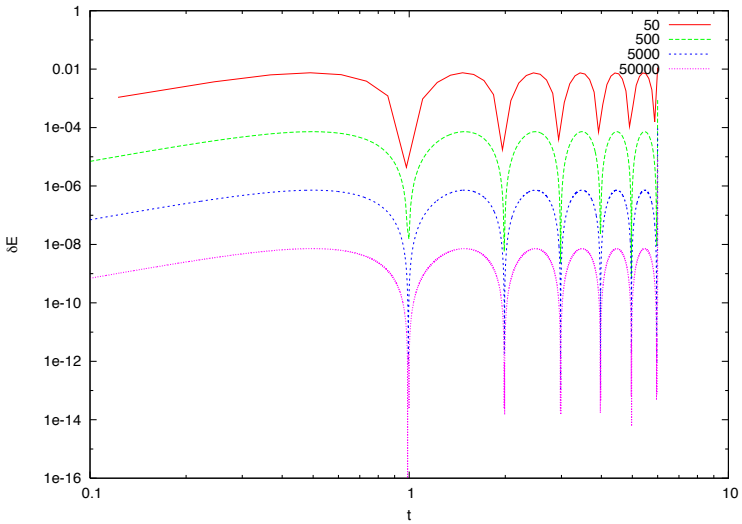



Figure 4.17: Like in figure 4.11 for the case of mechanical energy for the Euler–Verlet method.

all subprograms that declare a common block with the same name using a COMMON declaration. Such a declaration is shown in the following lines

```
 real(8) ::      omega_0,omega,gamma,a_0,omega_02,omega2
 common /params/omega_0,omega,gamma,a_0,omega_02,omega2
```

which when written in a (sub)program, the (sub)program gains access to the "memory position" `params` where the values of the variables are stored. Another point that needs our attention is the function `f2(t,x1,x2)` which now takes the velocity $v \to$ `x2` in its arguments:

```
real(8) function f2(t,x1,x2)
 implicit none
 real(8)         omega_0,omega,gamma,a_0,omega_02,omega2
 common /params/omega_0,omega,gamma,a_0,omega_02,omega2
 real(8) t,x1,x2,a
 a = a_0*cos(omega*t)
 f2=-omega_02*x1-gamma*x2+a
end function f2
```

The main program found in the file `dlo.f90` is listed below. The subroutines `RK`, `RKSTEP` are the same as in `rk.f90` and should also be included in the same file.

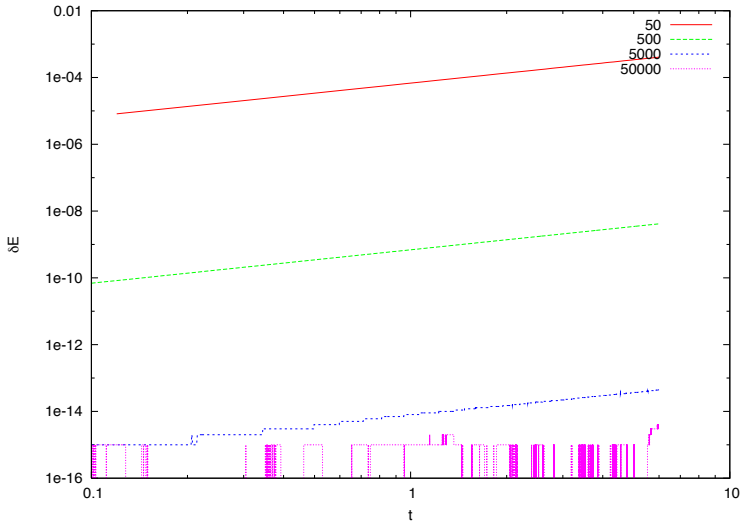



Figure 4.18: Like in figure 4.11 for the case of mechanical energy for the 4th order Runge–Kutta method. Roundoff errors appear for large enough number of steps.

```
!=========================================================
!Program to solve Damped Linear Oscillator
!using 4th order Runge-Kutta Method
!Output is written in file dlo.dat
!=========================================================
program dlo_solve
 implicit none
 integer, parameter  :: P=110000
 real(8),dimension(P):: T,X1,X2
 real(8) :: Ti,Tf,X10,X20
 real(8) :: Energy
 real(8) ::     omega_0,omega,gamma,a_0,omega_02,omega2
 common /params/omega_0,omega,gamma,a_0,omega_02,omega2
 integer :: Nt, i
!Input:
 print *,'Runge-Kutta Method for DLO Integration'
 print *,'Enter omega_0, omega, gamma, a_0:'
 read  *, omega_0,omega,gamma,a_0
 omega_02 = omega_0*omega_0
 omega2   = omega  *omega
 print *, 'omega_0= ',omega_0,' omega= ', omega
 print *, 'gamma=   ',gamma,   ' a_0=   ',a_0
 print *,'Enter Nt,Ti,TF,X10,X20:'
 read  *, Nt,Ti,Tf,X10,X20
 print *,'Nt = ',Nt
 print *,'Time: Initial Ti =',Ti,' Final Tf=',Tf
```

```
 print *,'            X1(Ti)=',X10,' X2(Ti)=',X20
 if(Nt.gt.P) stop 'Nt>P'
!The Calculation:
 call RK(T,X1,X2,Ti,Tf,X10,X20,Nt)
!Output:
 open(unit=11,file='dlo.dat')
 write(11,*)'# Damped Linear Oscillator - dlo'
 write(11,*)'# omega_0= ',omega_0,' omega= ', omega,&
      ' gamma= ',gamma,' a_0= ',a_0
 do i=1,Nt
  Energy = 0.5D0*X2(i)*X2(i)+0.5D0*omega_02*X1(i)*X1(i)
  write(11,*)T(i),X1(i),X2(i),Energy
 enddo
 close(11)
end program dlo_solve
!==========================================================
!The functions f1,f2(t,x1,x2) provided by the user
!==========================================================
real(8) function f1(t,x1,x2)
 implicit none
 real(8) t,x1,x2
 f1=x2            !dx1/dt= v = x2
end function f1
!----------------------------------------------------------
real(8) function f2(t,x1,x2)
 implicit none
 real(8)        omega_0,omega,gamma,a_0,omega_02,omega2
 common /params/omega_0,omega,gamma,a_0,omega_02,omega2
 real(8) t,x1,x2,a
 a = a_0*cos(omega*t)
 f2=-omega_02*x1-gamma*x2+a
end function f2
```

The results are shown in figures 4.19–4.22. Figure 4.19 shows the transition from a damped motion for $\gamma > 2\omega_0$ to an oscillating motion with damping amplitude for $\gamma < 2\omega_0$. The exponential decrease of the amplitude is shown in figure 4.21, whereas the dependence of the period $T$ from the damping coefficient $\gamma$ is shown in figure 4.22. Motivated by equation (4.28), written in the form

$$4\omega_0^2 - \left(\frac{2\pi}{T}\right) = \gamma^2 , \tag{4.32}$$

we construct the plot in figure 4.22. The right hand side of the equation is put on the horizontal axis, whereas the left hand side on the vertical. Equation (4.32) predicts that both quantities are equal and all measurements should lie on a particular line, the diagonal $y = x$. The period $T$

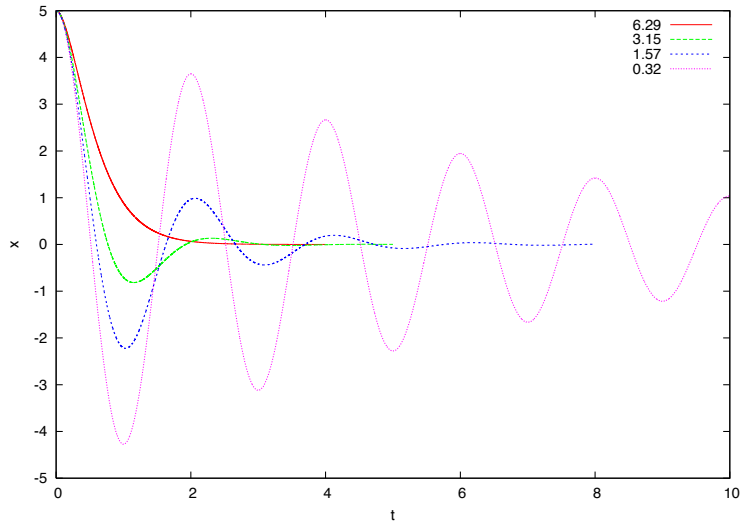



Figure 4.19: The position as a function of time for the damped oscillator for several values of $\gamma$ and $\omega_0 = 3.145$.

can be estimated from the time between two consecutive extrema of $x(t)$ or two consecutive zeros of the velocity $v(t)$ (see figure 4.19).

Finally it is important to study the trajectory of the system in phase space. This can be seen[10] in figure 4.20. A point in this space is a *state* of the system and a trajectory describes the evolution of the system's states in time. We see that all such trajectories end up as $t \to +\infty$ to the point $(0, 0)$, independently of the initial conditions. Such a point is an example of a system's *attractor*.

Next, we add the external force and study the response of the system to it. The system exhibits a *transient behavior* that depends on the initial conditions. For large enough times it approaches a *steady state* that does not depend on (almost all of) the initial conditions. This can be seen in figure 4.23. This is easily understood for our system by looking at equations (4.26)–(4.28). We see that the steady state $x_s(t)$ becomes dominant when the exponentials have damped away. $x_s(t)$ can be written in the form

$$\begin{aligned} x(t) &= x_0(\omega)\cos(\omega t + \delta(\omega)) \\ x_0(\omega) &= \frac{a_0}{\sqrt{(\omega_0^2 - \omega^2)^2 + \gamma^2\omega^2}}\,, \qquad \tan\delta(\omega) = \frac{\omega\gamma}{\omega^2 - \omega_0^2}\,. \end{aligned} \tag{4.33}$$

[10]To be precise, phase space is the space of positions-momenta, but in our case the difference is trivial.

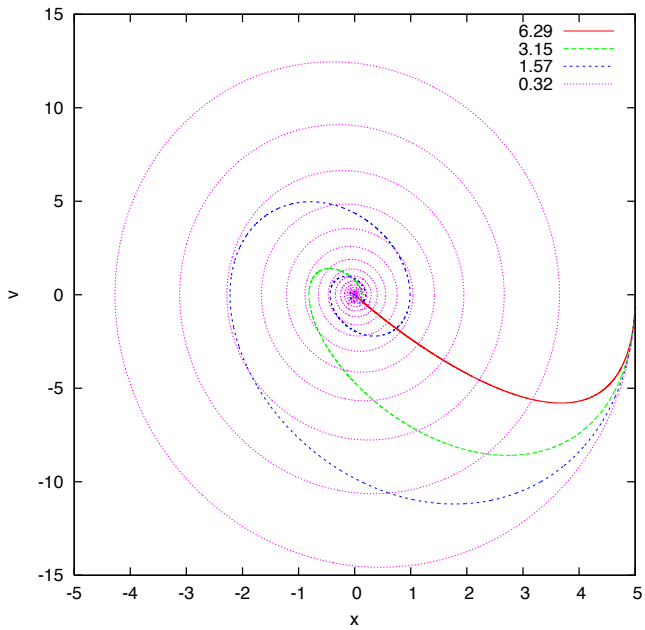



Figure 4.20: The phase space trajectory for the damped oscillator for several values of $\gamma$ and $\omega_0 = 3.145$. Note the attractor at $(x, v) = (0, 0)$ where all trajectories are "attracted to" as $t \to +\infty$.

These equations are verified in figure 4.24 where we study the dependence of the amplitude $x_0(\omega)$ on the angular frequency of the driving force. Finally we study the trajectory of the system in phase space. As we can see in figure 4.20, this time the attractor is an ellipse, which is a one dimensional curve instead of a zero dimensional point. For large enough times, all trajectories approach their attractor asymptotically.

## 4.6 The Forced Damped Pendulum

In this section we will study a non-linear dynamical system which exhibits interesting chaotic behavior. This is a simple model which, despite its deterministic nature, the prediction of its future behavior becomes intractable after a short period of time. Consider a simple pendulum in a constant gravitational field whose motion is damped by a force proportional to its velocity and it is under the influence of a vertical, harmonic external driving force:

$$\frac{d^2\theta}{dt^2} + \gamma\frac{d\theta}{dt} + \omega_0^2 \sin\theta = -2A\cos\omega t\ \sin\theta\,. \tag{4.34}$$

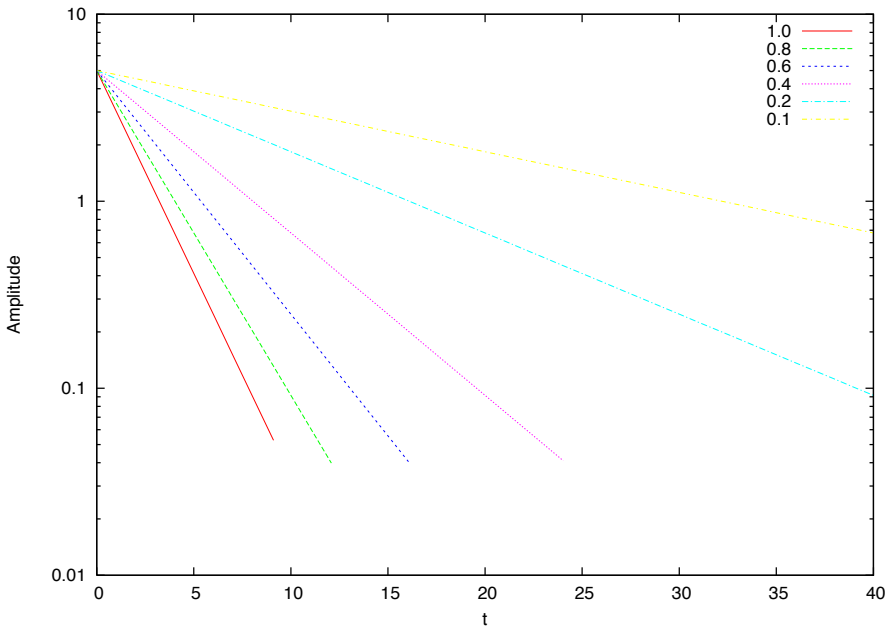



Figure 4.21: The amplitude of oscillation for the damped oscillator for several values of $\gamma$ and $\omega_0 = 3.145$. Note the exponential damping of the amplitude with time.

In the equation above, $\theta$ is the angle of the pendulum with the vertical axis, $\gamma$ is the damping coefficient, $\omega_0^2 = g/L$ is the pendulum's natural angular frequency, $\omega$ is the angular frequency of the driving force and $2A$ is the amplitude of the external angular acceleration caused by the driving force.

In the absence of the driving force, the damping coefficient drives the system to the point $(\theta, \dot{\theta}) = (0, 0)$, which is an attractor for the system. This continues to happen for small enough $A$, but for $A > A_c$ the behavior of the system becomes more complicated.

The program that integrates the equations of motion of the system can be obtained by making trivial changes to the program in the file `dlo.f90`. This changes are listed in detail below, but we note that `X1` $\leftrightarrow \theta$, `X2` $\leftrightarrow \dot{\theta}$, `a_0` $\leftrightarrow A$. The final program can be found in the file `fdp.f90`. It is listed below, with the understanding that the commands in between the dots are the same as in the programs found in the files `dlo.f90, rk.f90`.

```
!=========================================================
!Program to solve Forced Damped Pendulum
!using 4th order Runge-Kutta Method
```

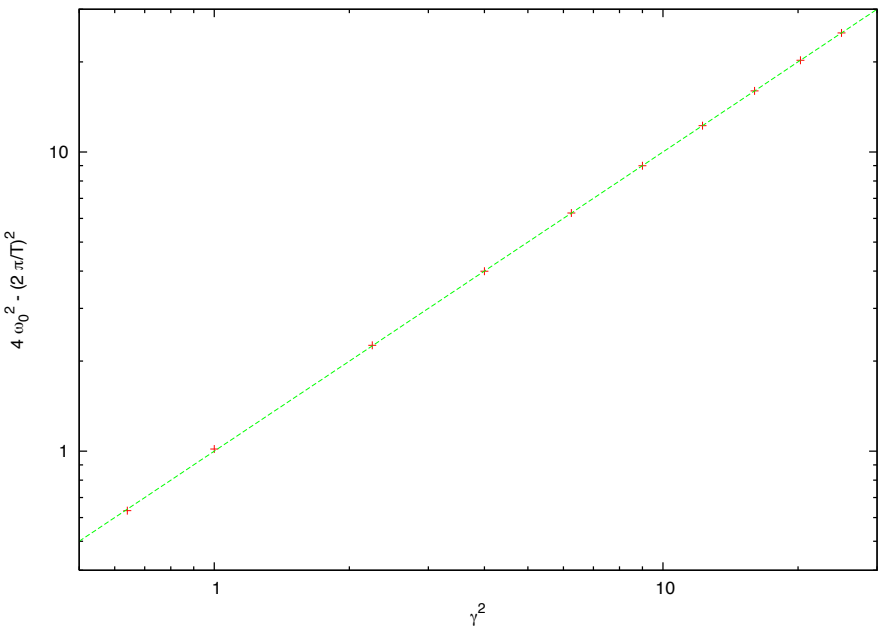



Figure 4.22: The period of oscillation of the damped oscillator for several values of $\gamma$ and $\omega_0 = 3.145$. The axes are chosen so that equation (4.28) $(2\pi/T)^2 = 4\omega_0^2 - \gamma^2$ can be easily verified. The points in the plot are our measurements whereas the straight line is the theoretical prediction, the diagonal $y = x$

```
!Output is written in file fdp.dat
!=========================================================
program dlo_solve
 implicit none
 integer, parameter  :: P=1010000
 ..............................
  Energy = 0.5D0*X2(i)*X2(i)+omega_02*(1.0D0-cos(X1(i)))
 ..............................
end program dlo_solve
!---------------------------------------------------------
real(8) function f2(t,x1,x2)
 implicit none
 real(8)        omega_0,omega,gamma,a_0,omega_02,omega2
 common /params/omega_0,omega,gamma,a_0,omega_02,omega2
 real(8) t,x1,x2
 f2=-(omega_02+2.0D0*a_0*cos(omega*t))*sin(x1)-gamma*x2
end function f2
!=========================================================
subroutine RKSTEP(t,x1,x2,dt)
 implicit none
 ..............................
```

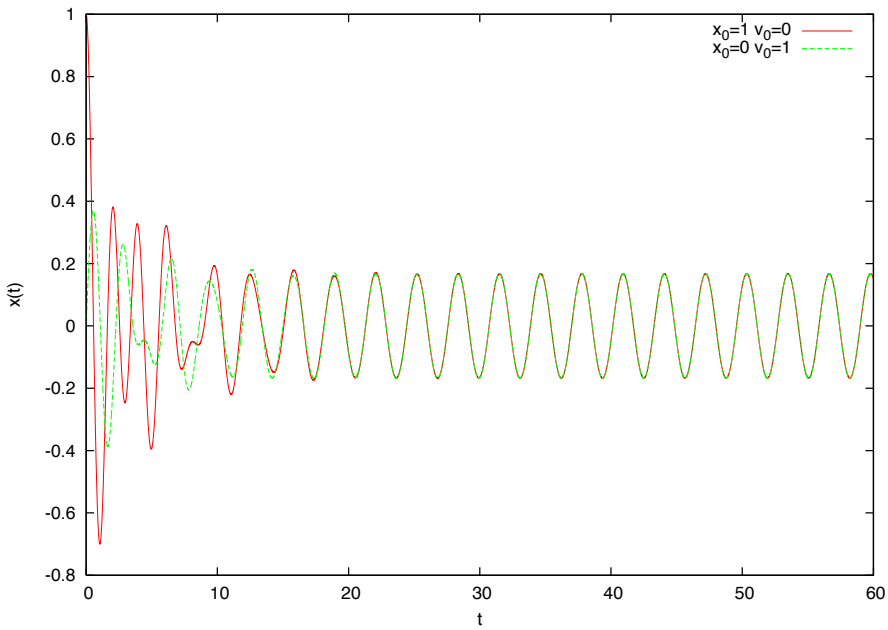



Figure 4.23: The period of oscillation for the forced damped oscillator for different initial conditions. We have chosen $\omega_0 = 3.145$, $\omega = 2.0$, $\gamma = 0.5$ and $a_0 = 1.0$. We note that after the transient behavior the system oscillates harmonically according to the relation $x(t) = x_0(\omega)\cos(\omega t + \delta)$.

```
 real(8),parameter :: pi =3.14159265358979324D0
 real(8),parameter :: pi2=6.28318530717958648D0
 ..............................
 x1 =x1+h6*(k11+2.0D0*(k12+k13)+k14)
 x2 =x2+h6*(k21+2.0D0*(k22+k23)+k24)
 if( x1 .gt.  pi) x1 = x1 - pi2
 if( x1 .lt. -pi) x1 = x1 + pi2
end subroutine RKSTEP
```

The final lines in the program are added so that the angle is kept within the interval $[-\pi, \pi]$.

In order to study the system's properties we will set $\omega_0 = 1$, $\omega = 2$, and $\gamma = 0.2$ unless we explicitly state otherwise. The natural period of the pendulum is $T_0 = 2\pi/\omega_0 = 2\pi \approx 6.28318530717958648$ whereas that of the driving force is $T = 2\pi/\omega = \pi \approx 3.14159265358979324$. For $A < A_c$, with $A_c \approx 0.18$, the point $(\theta, \dot{\theta}) = (0, 0)$ is an attractor, which means that the pendulum eventually stops at its stable equilibrium point. For $A_c < A < 0.71$ the attractor is a closed curve, which means that the pendulum at its steady state oscillates indefinitely without circling

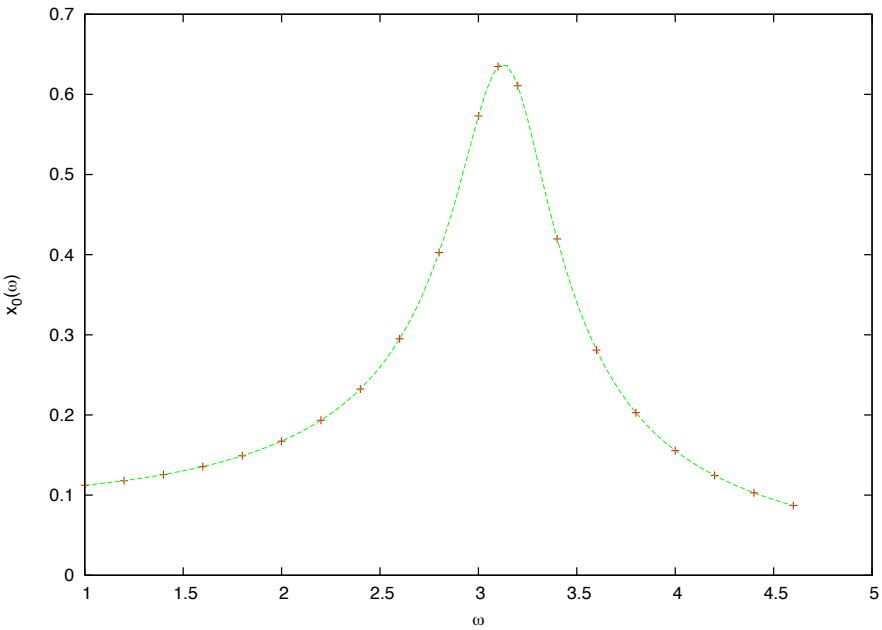



Figure 4.24: The oscillation amplitude $x_0(\omega)$ as a function of $\omega$ for the forced damped oscillator, where $\omega_0 = 3.145$, $\gamma = 0.5$ and $a_0 = 1.0$. We observe a resonance for $\omega \approx \omega_0$. The points of the plot are our measurements and the line is the theoretical prediction given by equation (4.33).

through its unstable equilibrium point at $\theta = \pm\pi$. The period of motion is found to be twice that of the driving force. For $0.72 < A < 0.79$ the attractor is an open curve, because at its steady state the pendulum crosses the $\theta = \pm\pi$ point. The period of the motion becomes equal to that of the driving force. For $0.79 < A \lesssim 1.033$ we have *period doubling* for critical values of $A$, but the trajectory is still periodic. For even larger values of $A$ the system enters into a chaotic regime where the trajectories are non periodic. For $A \approx 3.1$ we find the system in a periodic steady state again, whereas for $A \approx 3.8 - 4.448$ we have period doubling. For $A \approx 4.4489$ we enter into a chaotic regime again etc. These results can be seen in figures 4.27–4.29. The reader should construct the bifurcation diagram of the system by solving problem 20 of this chapter.

We can also use the so called *Poincaré* diagrams in order to study the chaotic behavior of a system. These are obtained by placing a point in phase space when the time is an integer multiple of the period of the driving force. Then, if for example the period of the motion is equal to that of the period of the driving force, the Poincaré diagram consists of only one point. If the period of the motion is an $n$–multiple of the

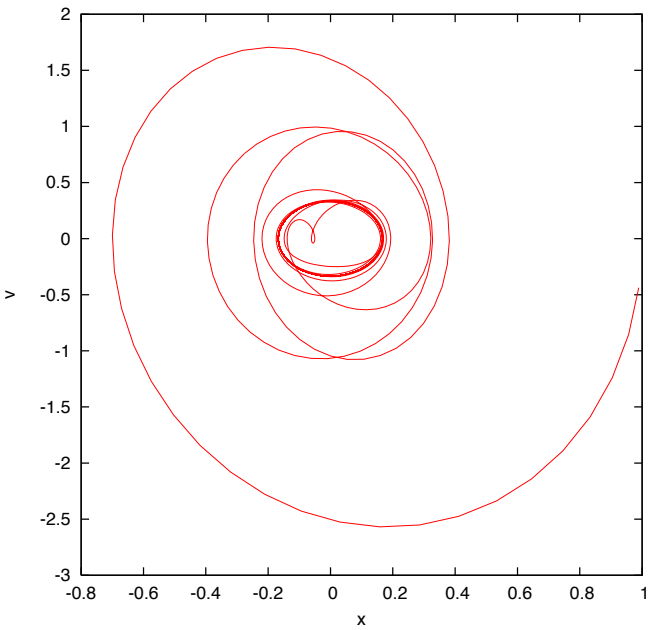



Figure 4.25: A phase space trajectory of the forced damped oscillator with $\omega_0 = 3.145$, $\omega = 2.0$, $\gamma = 0.5$ and $a_0 = 1.0$. The harmonic oscillation which is the steady state of the system is an ellipse, which is an attractor of all the phase space trajectories that correspond to different initial conditions.

period of the driving force then the Poincaré diagram consists of only $n$ points. Therefore, in the period doubling regime, the points of the Poincaré diagram double at each period doubling point. In the chaotic regime, the Poincaré diagram consists of an infinite number of points which belong to sets that have interesting fractal structure. One way to construct the Poincaré diagram numerically, is to process the data of the output file `fdp.dat` using `awk`[11]:

```
awk -v o=$omega -v nt=$Nt -v tf=$TF \
 'BEGIN{T=6.283185307179/o;dt=tf/nt;} $1%T<dt{print $2,$3}'\
 fdp.dat
```

where `$omega`, `$Nt`, `$TF` are the values of the angular frequency $\omega$, the number of points of time and the final time $t_f$. We calculate the period `T` and the time step `dt` in the program. Then we print those lines of the file

[11]The command can be written in one line without the final \ of the first and second lines.

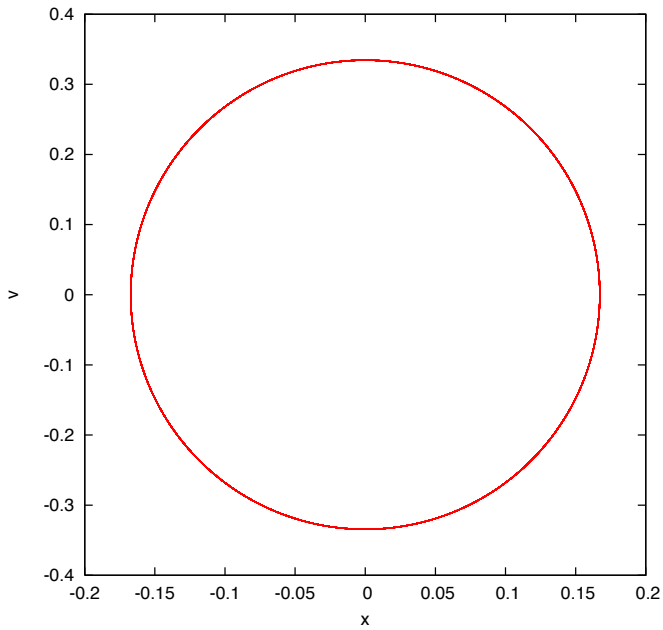



Figure 4.26: The trajectory shown in figure 4.25 for $t > 100$. The trajectory is almost on top of an ellipse corresponding to the steady state motion of the system. This ellipse is an attractor of the system.

where the time is an integer multiple of the period[12]. This is accomplished by the modulo operation `$1 % T`. The value of the expression `$1 % T < dt` is true when the remainder of the division of the first column (`$1`) of the file `fdp.dat` with the period `T` is smaller than `dt`. The results in the chaotic regime are displayed in figure 4.30.

We close this section by discussing another concept that helps us in the analysis of the dynamical properties of the pendulum. This is the concept of the *basin of attraction* which is the set of initial conditions in phase space that lead the system to a specific attractor. Take for example the case for $A > 0.79$ in the regime where the pendulum at its steady state has a circular trajectory with a positive or negative direction. By taking a large sample of initial conditions and recording the direction of the resulting motion after the transient behavior, we obtain figure 4.31.

[12]The accuracy of this condition is limited by `dt`, which makes the points in the Poincaré diagram slightly fuzzy.

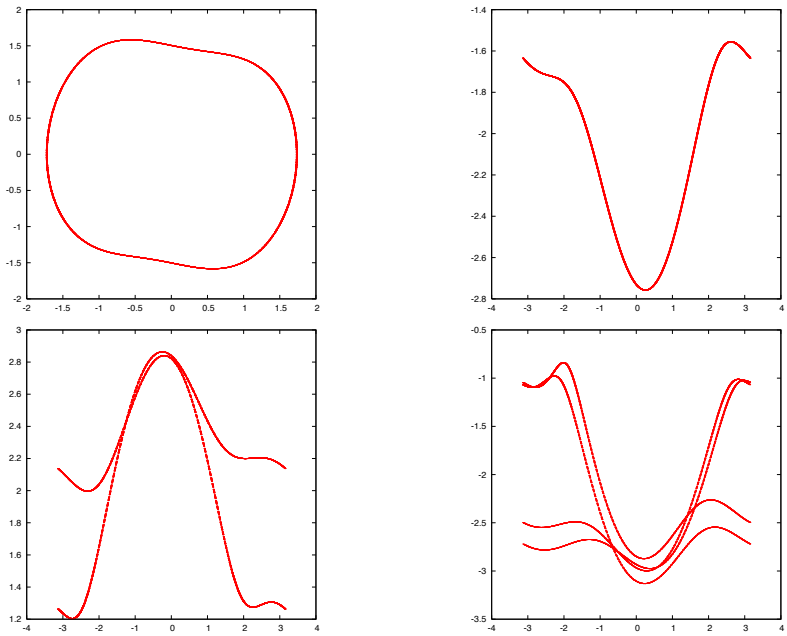


Figure 4.27: A phase space trajectory of the forced damped pendulum. The parameters chosen are $\omega_0 = 1.0$, $\omega = 2.0$, $\gamma = 0.2$ and $A = 0.60, 0.72, 0.85, 1.02$. We observe the phenomenon of period doubling.

## 4.7 Appendix: On the Euler–Verlet Method

Equations (4.11) can be obtained from the Taylor expansion

$$\begin{aligned}
\theta(t+\Delta t) &= \theta(t) + (\Delta t)\theta'(t) + \frac{(\Delta t)^2}{2!}\theta''(t) + \frac{(\Delta t)^3}{3!}\theta'''(t) + \mathcal{O}((\Delta t)^4) \\
\theta(t-\Delta t) &= \theta(t) - (\Delta t)\theta'(t) + \frac{(\Delta t)^2}{2!}\theta''(t) - \frac{(\Delta t)^3}{3!}\theta'''(t) + \mathcal{O}((\Delta t)^4)\,.
\end{aligned}$$

By adding and subtracting the above equations we obtain

$$\begin{aligned}
\theta(t+\Delta t) + \theta(t-\Delta t) &= 2\theta(t) + (\Delta t)^2\theta''(t) + \mathcal{O}((\Delta t)^4) \\
\theta(t+\Delta t) - \theta(t-\Delta t) &= 2(\Delta t)\theta'(t) + \mathcal{O}((\Delta t)^3)
\end{aligned} \tag{4.35}$$

which give equations (4.11)

$$\begin{aligned}
\theta(t+\Delta t) &= 2\theta(t) - \theta(t-\Delta t) + (\Delta t)^2\alpha(t) + \mathcal{O}((\Delta t)^4) \\
\omega(t) &= \frac{\theta(t+\Delta t) - \theta(t-\Delta t)}{2(\Delta t)} + \mathcal{O}((\Delta t)^2)
\end{aligned} \tag{4.36}$$

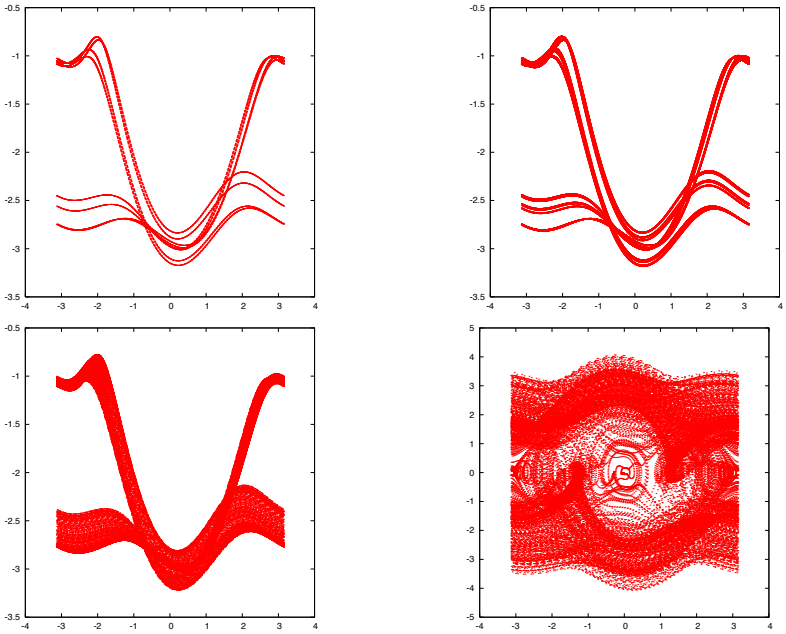


Figure 4.28: A phase space trajectory of the forced damped pendulum. The parameters chosen are $\omega_0 = 1.0$, $\omega = 2.0$, $\gamma = 0.2$ and $A = 1.031, 1.033, 1.04, 1.4$. We observe the chaotic behavior of the system.

From the first equation and equations (4.9) we obtain:

$$\theta(t + \Delta t) = \theta(t) + \omega(t)(\Delta t) + \mathcal{O}((\Delta t)^2) \tag{4.37}$$

When we perform a numerical integration, we are interested in the total error accumulated after $N - 1$ integration steps. In this method, these errors must be studied carefully:

- The error in the velocity $\omega(t)$ does not accumulate because it is given by the *difference* of the positions $\theta(t + \Delta t) - \theta(t - \Delta t)$.

- The accumulation of the errors for the position is estimated as follows: Assume that $\delta\theta(t)$ is the *total* accumulated error from the integration from time $t_0$ to $t$. Then according to the expansions

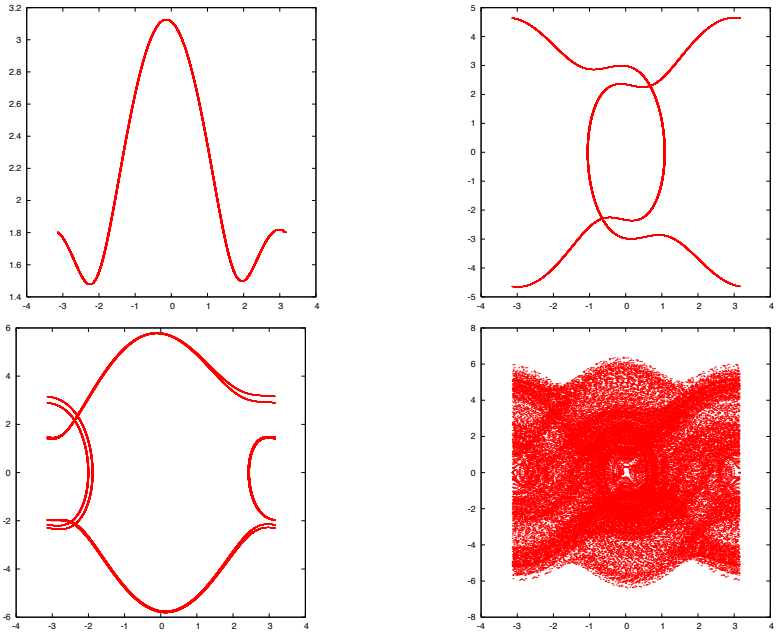


Figure 4.29: A phase space trajectory of the forced damped pendulum. The parameters chosen are $\omega_0 = 1.0$, $\omega = 2.0$, $\gamma = 0.2$ and $A = 1.568, 3.8, 4.44, 4.5$. We observe the system exiting and reentering regimes of chaotic behavior.

(4.36) the error for the first step is $\delta\theta(t_0 + \Delta t) = \mathcal{O}((\Delta t)^4)$. Then[13]

$$\begin{aligned}
\theta(t_0 + 2\Delta t) &= 2\theta(t_0 + \Delta t) - \theta(t_0) + \Delta t^2 \alpha(t_0 + \Delta t) + \mathcal{O}((\Delta t)^4) \Rightarrow \\
\delta\theta(t_0 + 2\Delta t) &= 2\delta\theta(t_0 + \Delta t) - \delta\theta(t_0) + \mathcal{O}((\Delta t)^4) \\
&= 2\mathcal{O}((\Delta t)^4) - 0 + \mathcal{O}((\Delta t)^4) \\
&= 3\mathcal{O}((\Delta t)^4)\,.
\end{aligned}$$

For the next steps we obtain

$$\begin{aligned}
\theta(t_0 + 3\Delta t) &= 2\theta(t_0 + 2\Delta t) - \theta(t_0 + \Delta t) + \Delta t^2 \alpha(t_0 + 2\Delta t) + \mathcal{O}((\Delta t)^4) \Rightarrow \\
\delta\theta(t_0 + 3\Delta t) &= 2\delta\theta(t_0 + 2\Delta t) - \delta\theta(t_0 + \Delta t) + \mathcal{O}((\Delta t)^4) \\
&= 6\mathcal{O}((\Delta t)^4) - \mathcal{O}((\Delta t)^4) + \mathcal{O}((\Delta t)^4) \\
&= 6\mathcal{O}((\Delta t)^4)\,,
\end{aligned}$$

[13]Remember that the acceleration $\alpha(t)$ is given, therefore $\delta\alpha(t) = 0$.

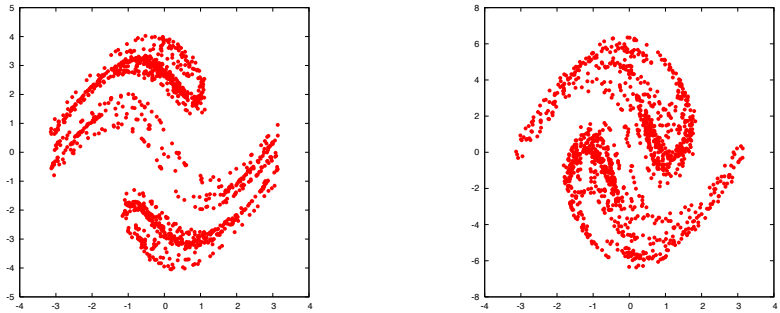


Figure 4.30: A Poincaré diagram for the forced damped pendulum in its chaotic regime. The parameters chosen are $\omega_0 = 1.0$, $\omega = 2.0$, $\gamma = 0.2$ and $A = 1.4, 4.5$.

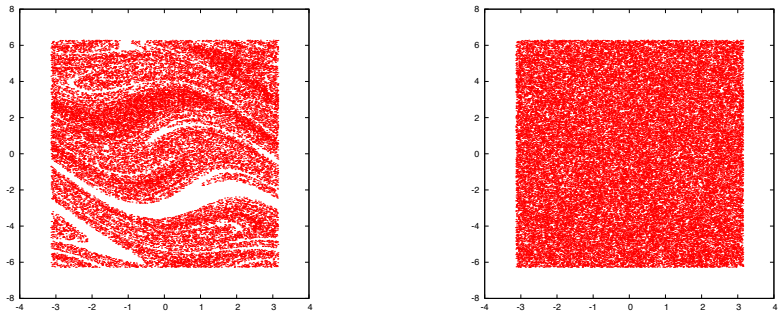


Figure 4.31: Basin of attraction for the forced damped pendulum. The parameters chosen are $\omega_0 = 1.0$, $\omega = 2.0$, $\gamma = 0.2$ and $A = 0.85, 1.4$.

$$
\begin{aligned}
\theta(t_0 + 4\Delta t) &= 2\theta(t_0 + 3\Delta t) - \theta(t_0 + 2\Delta t) + \Delta t^2 \alpha(t_0 + 3\Delta t) + \mathcal{O}((\Delta t)^4) \Rightarrow \\
\delta\theta(t_0 + 4\Delta t) &= 2\delta\theta(t_0 + 3\Delta t) - \delta\theta(t_0 + 2\Delta t) + \mathcal{O}((\Delta t)^4) \\
&= 12\mathcal{O}((\Delta t)^4) - 3\mathcal{O}((\Delta t)^4) + \mathcal{O}((\Delta t)^4) \\
&= 10\mathcal{O}((\Delta t)^4)\,.
\end{aligned}
$$

Then, inductively, if $\delta\theta(t_0 + (n-1)\Delta t) = \frac{(n-1)n}{2}\mathcal{O}((\Delta t)^4)$, we obtain

$$
\begin{aligned}
\theta(t_0 + n\Delta t) &= 2\theta(t_0 + (n-1)\Delta t) - \theta(t_0 + (n-2)\Delta t) + \Delta t^2 \alpha(t_0 + (n-1)\Delta t) \\
&\quad + \mathcal{O}((\Delta t)^4) \Rightarrow \\
\delta\theta(t_0 + n\Delta t) &= 2\delta\theta(t_0 + (n-1)\Delta t) - \delta\theta(t_0 + (n-2)\Delta t) + \mathcal{O}((\Delta t)^4) \\
&= 2\frac{(n-1)n}{2}\mathcal{O}((\Delta t)^4) - \frac{(n-2)(n-1)}{2}\mathcal{O}((\Delta t)^4) + \mathcal{O}((\Delta t)^4) \\
&= \frac{n(n+1)}{2}\mathcal{O}((\Delta t)^4)\,.
\end{aligned}
$$

Finally

$$\delta\theta(t_0 + n\Delta t) = \frac{n(n+1)}{2}\mathcal{O}((\Delta t)^4) \sim \frac{1}{\Delta t^2}\mathcal{O}((\Delta t)^4) \sim \mathcal{O}((\Delta t)^2)\,. \tag{4.38}$$

Therefore the total error is $\mathcal{O}((\Delta t)^2)$.

We also mention the Velocity Verlet method or the Leapfrog method. In this case we use the velocity explicitly:

$$\begin{aligned} \theta_{n+1} &= \theta_n + \omega_n \Delta t + \frac{1}{2}\alpha_n \Delta t^2 \\ \omega_{n+\frac{1}{2}} &= \omega_n + \frac{1}{2}\alpha_n \Delta t \\ \omega_{n+1} &= \omega_{n+\frac{1}{2}} + \frac{1}{2}\alpha_{n+1}\Delta t\,. \end{aligned} \tag{4.39}$$

The last step uses the acceleration $\alpha_{n+1}$ which should depend only on the position $\theta_{n+1}$ and not on the velocity.

The Verlet methods are popular in *molecular dynamics* simulations of many body systems. One of their advantages is that the constraints of the system of particles are easily encoded in the algorithm.

## 4.8 Appendix: 2nd order Runge–Kutta Method

In this appendix we will show how the choice of the intermediate point 2 in equation (4.17) reduces the error by a power of $h$. This choice is special, since by choosing another point (e.g. $t = t_n + 0.4h$) the result would have not been the same. Indeed, from the relation

$$\frac{dx}{dt} = f(t,x) \Rightarrow x_{n+1} = x_n + \int_{t_n}^{t_{n+1}} f(t,x)\,dx\,. \tag{4.40}$$

By Taylor expanding around the point $(t_{n+1/2}, x_{n+1/2})$ we obtain

$$f(t,x) = f(t_{n+1/2}, x_{n+1/2}) + (t - t_{n+1/2})\frac{df}{dt}(t_{n+1/2}) + \mathcal{O}(h^2)\,. \tag{4.41}$$

Therefore

$$
\begin{aligned}
&\int_{t_n}^{t_{n+1}} f(t,x)\,dx \\
&= f(t_{n+1/2}, x_{n+1/2})(t_{n+1}-t_n) + \frac{df}{dt}(t_{n+1/2})\,\frac{(t-t_{n+1/2})^2}{2}\bigg|_{t_n}^{t_{n+1}} \\
&\quad +\mathcal{O}(h^2)(t_{n+1}-t_n) \\
&= f(t_{n+1/2}, x_{n+1/2})h + \frac{df}{dt}(t_{n+1/2})\left\{\frac{(t_{n+1}-t_{n+1/2})^2}{2} - \frac{(t_n-t_{n+1/2})^2}{2}\right\} \\
&\quad +\mathcal{O}(h^2)h \\
&= f(t_{n+1/2}, x_{n+1/2})h + \frac{df}{dt}(t_{n+1/2})\left\{\frac{h^2}{2} - \frac{(-h)^2}{2}\right\} + \mathcal{O}(h^3) \\
&= f(t_{n+1/2}, x_{n+1/2})h + \mathcal{O}(h^3)\,. \qquad (4.42)
\end{aligned}
$$

Note that for the vanishing of the $\mathcal{O}(h)$ term it is necessary to place the intermediate point at time $t_{n+1/2}$.

This is not a unique choice. This can be most easily seen by a different analysis of the Taylor expansion. Expanding around the point $(t_n, x_n)$ we obtain

$$
\begin{aligned}
x_{n+1} &= x_n + (t_{n+1}-t_n)\frac{dx_n}{dt} + \frac{1}{2}(t_{n+1}-t_n)^2\frac{d^2x_n}{dt^2} + \mathcal{O}(h^3) \\
&= x_n + hf_n + \frac{h^2}{2}\frac{df_n}{dt} + \mathcal{O}(h^3) \\
&= x_n + hf_n + \frac{h^2}{2}\left(\frac{\partial f_n}{\partial t} + \frac{\partial f_n}{\partial x}\frac{dx_n}{dt}\right) + \mathcal{O}(h^3) \\
&= x_n + hf_n + \frac{h^2}{2}\left(\frac{\partial f_n}{\partial t} + \frac{\partial f_n}{\partial x}f_n\right) + \mathcal{O}(h^3)\,, \qquad (4.43)
\end{aligned}
$$

where we have set $f_n \equiv f(t_n, x_n)$, $\frac{dx_n}{dt} \equiv \frac{dx}{dt}(x_n)$ etc. We define

$$
\begin{aligned}
k_1 &= f(t_n, x_n) = f_n \\
k_2 &= f(t_n + ah, x_n + bhk_1) \\
x_{n+1} &= x_n + h(c_1k_1 + c_2k_2)\,. \qquad (4.44)
\end{aligned}
$$

and we will determine the conditions so that the terms $\mathcal{O}(h^2)$ of the last equation in the error are identical with those of equation (4.43). By

expanding $k_2$ we obtain

$$\begin{aligned}
k_2 &= f(t_n + ah, x_n + bhk_1) \\
&= f(t_n, x_n + bhk_1) + ha\frac{\partial f}{\partial t}(t_n, x_n + bhk_1) + \mathcal{O}(h^2) \\
&= f(t_n, x_n) + hbk_1\frac{\partial f}{\partial x}(t_n, x_n) + ha\frac{\partial f}{\partial t}(t_n, x_n) + \mathcal{O}(h^2) \\
&= f_n + h\left\{a\frac{\partial f_n}{\partial t} + bk_1\frac{\partial f_n}{\partial x}\right\} + \mathcal{O}(h^2) \\
&= f_n + h\left\{a\frac{\partial f_n}{\partial t} + bf_n\frac{\partial f_n}{\partial x}\right\} + \mathcal{O}(h^2) \qquad (4.45)
\end{aligned}$$

Substituting in (4.44) we obtain

$$\begin{aligned}
x_{n+1} &= x_n + h(c_1k_1 + c_2k_2) \\
&= x_n + h\left\{c_1f_n + c_2f_n + c_2h\left(a\frac{\partial f_n}{\partial t} + bf_n\frac{\partial f_n}{\partial x}\right) + \mathcal{O}(h^2)\right\} \\
&= x_n + h(c_1 + c_2)f_n + \frac{h^2}{2}\left((2c_2a)\frac{\partial f_n}{\partial t} + (2c_2b)f_n\frac{\partial f_n}{\partial x}\right) \\
&\quad + \mathcal{O}(h^3)\,. \qquad (4.46)
\end{aligned}$$

All we need is to choose

$$\begin{aligned}
c_1 + c_2 &= 1 \\
2c_2a &= 1 \\
2c_2b &= 1\,. \qquad (4.47)
\end{aligned}$$

The choice $c_1 = 0$, $c_2 = 1$, $a = b = 1/2$ leads to equation (4.19). Some other choices in the bibliography are $c_2 = 1/2$ and $c_2 = 3/4$.

## 4.9 Problems

4.1 Prove that the total error in the Euler–Cromer method is of order $\Delta t$.

4.2 Reproduce the results in figures 4.11–4.18

4.3 Improve your programs so that there is no accumulation of roundoff error in the calculation of time when `h` is very small for the methods Euler, Euler-Cromer, Euler-Verlet and Runge-Kutta. Repeat the analysis of the previous problem.

4.4 Make the appropriate changes in your programs of the Euler, Euler-Cromer, Euler-Verlet and Runge-Kutta methods so that all floating variables change from `REAL`→`REAL(8)`. Repeat the analysis of the previous problem.

4.5 Compare the results obtained from the Euler, Euler-Cromer, Euler-Verlet, Runge-Kutta methods for the following systems where the analytic solution is known:

(a) Particle falling in a constant gravitational field. Consider the case $v(0) = 0$, $m = 1$, $g = 10$.

(b) Particle falling in a constant gravitational field moving in a fluid from which exerts a force $F = -kv$ on the particle. Consider the case $v(0) = 0$, $m = 1$, $g = 10$ $k = 0.1, 1.0, 2.0$. Calculate the limiting velocity of the particle numerically and compare the value obtained to the theoretical expectation.

(c) Repeat for the case of a force of resistance of magnitude $|F| = kv^2$.

4.6 Consider the damped harmonic oscillator

$$\frac{d^2x}{dt^2} + \gamma\frac{dx}{dt} + \omega_0^2 x = 0\,. \tag{4.48}$$

Take $\omega_0 = 3.145$, $\gamma = 0.5$ and calculate its mechanical energy as a function of time. Is it monotonic? Why? (show that $d(E/m)/dt = -\gamma v^2$). Repeat for $\gamma = 4, 5, 6, 7, 8$. When is the system oscillating and when it's not? Calculate numerically the critical value of $\gamma$ for which the system passes from a non oscillating to an oscillating regime. Compare your results with the theoretical expectations.

4.7 Reproduce the results of figures 4.19–4.22.

4.8 Reproduce the results of figures 4.23–4.26. Calculate the phase $\delta(\omega)$ numerically and compare with equation (4.33).

4.9 Consider a simple model for a swing. Take the damped harmonic oscillator and a driving force which periodically exerts a momentary push with angular frequency $\omega$. Define "momentary" to be an impulse given by the acceleration $a_0$ by an appropriately small time interval $\Delta t$. The acceleration is 0 for all other times. Calculate the amplitude $x_0(\omega)$ for $\omega_0 = 3.145$ and $\gamma = 0.5$.

4.10 Consider a "half sine" driving force on a damped harmonic oscillator

$$a(t) = \begin{cases} a_0 \cos \omega t & \cos \omega t > 0 \\ 0 & \cos \omega t \leq 0 \end{cases}$$

Study the transient behavior of the system for several initial conditions and calculate its steady state motion for $\omega_0 = 3.145$ and $\gamma = 0.5$. Calculate the amplitude $x_0(\omega)$.

4.11 Consider the driving force on a damped oscillator given by

$$a(t) = \frac{1}{\pi} + \frac{1}{2} \cos \omega + \frac{2}{3\pi} \cos 2\omega t - \frac{2}{15\pi} \cos 4\omega t$$

Study the transient behavior of the system for several initial conditions and calculate its steady state motion for $\omega_0 = 3.145$ and $\gamma = 0.5$. Calculate the amplitude $x_0(\omega)$. Compare your results with those of the previous problem and comment about.

4.12 Write a program that simulates $N$ identical, independent harmonic oscillators. Take $N = 20$ and choose random initial conditions for each one of them. Study their trajectories in phase space and check whether they cross each other. Comment on your results.

4.13 Place the $N = 20$ harmonic oscillators of the previous problem in a small square in phase space whose center is at the origin of the axes. Consider the evolution of the system in time. Does the shape of the rectangle change in time? Does the area change in time? Explain...

4.14 Repeat the previous problem when each oscillator is damped with $\gamma = 0.5$. Take $\omega_0 = 3.145$.

4.15 Consider the forced damped oscillator with $\omega = 2$, $\omega_0 = 1.0$, $\gamma = 0.2$. Study the transient behavior of the system in the plots of $\theta(t)$, $\dot{\theta}(t)$ for $A = 0.1, 0.5, 0.79, 0.85, 1.03, 1.4$.

4.16 Consider the forced damped pendulum with $\omega = 2$, $\omega_0 = 1.0$, $\gamma = 0.2$ and study the phase space trajectories for $A =$ 0.1, 0.19, 0.21, 0.25, 0.5, 0.71, 0.79, 0.85, 1.02, 1.031, 1.033, 1.05, 1.08, 1.1, 1.4, 1.8, 3.1, 3.5, 3.8, 4.2, 4.42, 4.44, 4.445, 4.447, 4.4488. Consider both the transient behavior and the steady state motion.

4.17 Reproduce the results in figures 4.30.

4.18 Reproduce the results in figures 4.31.

4.19 Consider the forced damped oscillator with

$$\omega_0 = 1\,, \quad \omega = 2\,, \quad \gamma = 0.2$$

After the transient behavior, the motion of the system for $A = 0.60$, $A = 0.75$ and $A = 0.85$ is periodic. Measure the period of the motion with an accuracy of three significant digits and compare it with the natural period of the pendulum and with the period of the driving force. Take as initial conditions the following pairs: $(\theta_0, \dot{\theta}_0) = (3.1, 0.0)$, $(2.5, 0.0)$, $(2.0, 0.0)$, $(1.0, 0.0)$, $(0.2, 0.0)$, $(0.0, 1.0)$, $(0.0, 3.0)$, $(0.0, 6.0)$. Check if the period is independent of the initial conditions.

4.20 Consider the forced damped pendulum with

$$\omega_0 = 1\,, \quad \omega = 2\,, \quad \gamma = 0.2$$

Study the motion of the pendulum when the amplitude $A$ takes values in the interval $[0.2, 5.0]$. Consider specific discrete values of $A$ by splitting the interval above in subintervals of width equal to $\delta A = 0.002$. For each value of $A$, record in a file the value of $A$, the angular position and the angular velocity of the pendulum when $t_k = k\pi$ with $k = k_{trans}, k_{trans} + 1, k_{trans} + 2, \ldots, k_{max}$:

$$A \qquad \theta(t_k) \qquad \dot{\theta}(t_k)$$

The choice of $k_{trans}$ is made so that the transient behavior will be discarded and study only the steady state of the pendulum. You may take $k_{max} = 500$, $k_{trans} = 400$, $t_i = 0$, $t_f = 500\pi$, and split the intervals $[t_k, t_k + \pi]$ to 50 subintervals. Choose $\theta_0 = 3.1$, $\dot{\theta}_0 = 0$.

(a) Construct the bifurcation diagram by plotting the points $(A, \theta(t_k))$.

(b) Repeat by plotting the points $(A, \dot{\theta}(t_k))$.

(c) Check whether your results depend on the choice of $\theta_0$, $\dot{\theta}_0$. Repeat your analysis for $\theta_0 = 0$, $\dot{\theta}_0 = 1$.

(d) Study the onset of chaos: Take $A \in [1.0000, 1.0400]$ with $\delta A = 0.0001$ and $A \in [4.4300, 4.4500]$ with $\delta A = 0.0001$ and compute with the given accuracy the value $A_c$ where the system enters into the chaotic behavior regime.

(e) The plot the points $(\theta(t_k), \dot{\theta}(t_k))$ for $A = 1.034$, $1.040$, $1.080$, $1.400$, $4.450$, $4.600$. Put 2000 points for each value of $A$ and commend on the strength of the chaotic behavior of the pendulum.

# Chapter 5

# Planar Motion

In this chapter we will study the motion of a particle moving on the plane under the influence of a dynamical field. Special emphasis will be given to the study of the motion in a central field, like in the problem of planetary motion and scattering. We also study the motion of two or more interacting particles moving on the plane, which requires the solution of a larger number of dynamical equations. These problems can be solved numerically by using Runge–Kutta integration methods, therefore this chapter extends and applies the numerical methods studied in the previous chapter.

## 5.1 Runge–Kutta for Planar Motion

In two dimensions, the initial value problem that we are interested in, is solving the system of equations (4.6)

$$\begin{aligned} \frac{dx}{dt} &= v_x & \frac{dv_x}{dt} &= a_x(t, x, v_x, y, v_y) \\ \frac{dy}{dt} &= v_y & \frac{dv_y}{dt} &= a_y(t, x, v_x, y, v_y)\,. \end{aligned} \tag{5.1}$$

The 4th order Runge-Kutta method can be programmed by making small modifications of the program in the file `rk.f90`. In order to facilitate the study of many different dynamical fields, for each field we put the code of the respective acceleration in a different file. The code which is common for all the forces, namely the user interface and the implementation of the Runge–Kutta method, will be put in the file `rk2.f90`. The program that computes the acceleration will be put in a file named `rk_XXX.f90`, where `XXX` is a string of characters that identifies the force.

For example, the file `rk2_hoc.f90` contains the program computing the acceleration of the simple harmonic oscillator, the file `rk2_g.f90` the acceleration of a constant gravitational field $\vec{g} = -g\,\hat{y}$ etc.

Different force fields will require the use of one or more coupling constants which need to be accessible to the code in the main program and some subroutines. For this reason, we will provide two variables `k1, k2` in a `common block`:

```
real(8) ::          k1,k2
common /couplings/k1,k2
```

This common block will be accessed by the acceleration functions `f3` and `f4`, the function `energy` and the main program where the user will enter the values of `k1` and `k2`. The initial conditions are stored in the variables `X10` $\leftrightarrow x_0$, `X20` $\leftrightarrow y_0$, `V10` $\leftrightarrow v_{x0}$, `V20` $\leftrightarrow v_{y0}$, and the values of the functions of time will be stored in the arrays `X1(P)` $\leftrightarrow x(t)$, `X2(P)` $\leftrightarrow y(t)$, `V1(P)` $\leftrightarrow v_x(t)$, `V2(P)` $\leftrightarrow v_y(t)$. The integration is performed by a call to the subroutine

```
call RK(T,X1,X2,V1,V2,Ti,Tf,X10,X20,V10,V20,Nt)
```

The results are written to the file `rk2.dat`. Each line in this file contains the time, position, velocity and the total mechanical energy, where the energy is calculated by the function `energy(t,x1,x2,v1,v2)`:

```
open(unit=11,file='rk2.dat')
do i=1,Nt
 write(11,*)T(i),X1(i),X2(i),V1(i),V2(i),&
     energy(T(i),X1(i),X2(i),V1(i),V2(i))
enddo
```

The code for the function `energy`, which is different for each force field, is written in the same file with the acceleration. The code for the subroutine `RKSTEP(t,x1,x2,x3,x4,dt)` should be extended in order to integrate four instead of two functions. The full code is listed below:

```
!=========================================================
!Program to solve a 4 ODE system using Runge-Kutta Method
!User must supply derivatives
!dx1/dt=f1(t,x1,x2,x3,x4) dx2/dt=f2(t,x1,x2,x3,x4)
!dx3/dt=f3(t,x1,x2,x3,x4) dx4/dt=f4(t,x1,x2,x3,x4)
!as real(8) functions
```

```
!Output is written in file rk2.dat
!=========================================================
program rk2_solve
 implicit none
 integer,parameter    :: P=1010000
 real(8),dimension(P):: T,X1,X2,V1,V2
 real(8) :: Ti,Tf,X10,X20,V10,V20
 integer :: Nt, i
 real(8) ::         k1,k2
 common /couplings/k1,k2
 real(8) :: energy,E0,EF,DE
!Input:
 print *,'Runge-Kutta Method for 4-ODEs Integration'
 print *,'Enter coupling constants:'
 read  *, k1,k2
 print *,'k1= ',k1,' k2= ',k2
 print *,'Enter Nt,Ti,Tf,X10,X20,V10,V20:'
 read  *,       Nt,Ti,TF,X10,X20,V10,V20
 print *,'Nt = ',Nt
 print *,'Time: Initial Ti =',Ti,' Final Tf=',Tf
 print *,'            X1(Ti)=',X10,' X2(Ti)=',X20
 print *,'            V1(Ti)=',V10,' V2(Ti)=',V20
!The Calculation:
 call RK(T,X1,X2,V1,V2,Ti,Tf,X10,X20,V10,V20,Nt)
!Output:
 open(unit=11,file='rk2.dat')
 do i=1,Nt
  write(11,*)T(i),X1(i),X2(i),V1(i),V2(i),&
      energy(T(i),X1(i),X2(i),V1(i),V2(i))
 enddo
 close(11)
!Rutherford scattering angles:
 print *,'v-angle: ',atan2(V2(Nt),V1(Nt))
 print *,'b-angle: ',2.0D0*atan(k1/(V10*V10*X20))
 E0 = energy(Ti    ,X10    ,X20    ,V10    ,V20    )
 EF = energy(T(Nt),X1(Nt),X2(Nt),V1(Nt),V2(Nt))
 DE = ABS(0.5D0*(EF-E0)/(EF+E0))
 print *,'E0,EF, DE/E= ',E0,EF,DE
end program rk2_solve
!=========================================================
!The velocity functions f1,f2(t,x1,x2,v1,v2)
!=========================================================
real(8) function f1(t,x1,x2,v1,v2)
 implicit none
 real(8) :: t,x1,x2,v1,v2
 f1=v1            !dx1/dt= v1
end function f1
!---------------------------------------------------------
real(8) function f2(t,x1,x2,v1,v2)
```

```
 implicit none
 real(8) :: t,x1,x2,v1,v2
 f2=v2            !dx2/dt= v2
end function f2
!=========================================================
!RK(T,X1,X2,V1,V2,Ti,Tf,X10,X20,V10,V20,Nt) is the driver
!for the Runge-Kutta integration routine RKSTEP
!Input: Initial and final times Ti,Tf
!       Initial values at t=Ti  X10,X20,V10,V20
!       Number of steps of integration: Nt-1
!       Size of arrays T,X1,X2,V1,V2
!Output: real arrays T(Nt),X1(Nt),X2(Nt),
!                          V1(Nt),V2(Nt) where
!T(1) = Ti X1(1) = X10 X2(1) = X20 V1(1) = V10 V2(1) = V20
!          X1(k) = X1(at t=T(k)) X2(k) = X2(at t=T(k))
!          V1(k) = V1(at t=T(k)) V2(k) = V2(at t=T(k))
!T(Nt)= Tf
!=========================================================
subroutine RK(T,X1,X2,V1,V2,Ti,Tf,X10,X20,V10,V20,Nt)
 implicit none
 integer :: Nt
 real(8),dimension(Nt)::T,X1,X2,V1,V2
 real(8) :: Ti ,Tf
 real(8) :: X10,X20
 real(8) :: V10,V20
 real(8) :: dt
 real(8) :: TS,X1S,X2S !values of time and X1,X2 at given step
 real(8) ::    V1S,V2S
 integer :: i
!Initialize variables:
 dt     = (Tf-Ti)/(Nt-1)
 T (1)  = Ti
 X1(1)  = X10; X2(1) = X20
 V1(1)  = V10; V2(1) = V20
 TS     = Ti
 X1S    = X10; X2S   = X20
 V1S    = V10; V2S   = V20
!Make RK steps: The arguments of RKSTEP are
!replaced with the new ones
 do i=2,Nt
  call RKSTEP(TS,X1S,X2S,V1S,V2S,dt)
  T(i)  = TS
  X1(i) = X1S; X2(i) = X2S
  V1(i) = V1S; V2(i) = V2S
 enddo
end subroutine RK
!=========================================================
!Subroutine RKSTEP(t,x1,x2,dt)
!Runge-Kutta Integration routine of ODE
```

```
!dx1/dt=f1(t,x1,x2,x3,x4) dx2/dt=f2(t,x1,x2,x3,x4)
!dx3/dt=f3(t,x1,x2,x3,x4) dx4/dt=f4(t,x1,x2,x3,x4)
!User must supply derivative functions:
!real function f1(t,x1,x2,x3,x4)
!real function f2(t,x1,x2,x3,x4)
!real function f3(t,x1,x2,x3,x4)
!real function f4(t,x1,x2,x3,x4)
!Given initial point (t,x1,x2) the routine advances it
!by time dt.
!Input : Inital time t    and function values x1,x2,x3,x4
!Output: Final  time t+dt and function values x1,x2,x3,x4
!Careful!: values of t,x1,x2,x3,x4 are overwritten...
!=========================================================
subroutine RKSTEP(t,x1,x2,x3,x4,dt)
 implicit none
 real(8) :: t,x1,x2,x3,x4,dt
 real(8) :: f1,f2,f3,f4
 real(8) :: k11,k12,k13,k14,k21,k22,k23,k24
 real(8) :: k31,k32,k33,k34,k41,k42,k43,k44
 real(8) :: h,h2,h6

 h =dt        !h =dt, integration step
 h2=0.5D0*h   !h2=h/2
 h6=h/6.0D0   !h6=h/6

 k11=f1(t,x1,x2,x3,x4)
 k21=f2(t,x1,x2,x3,x4)
 k31=f3(t,x1,x2,x3,x4)
 k41=f4(t,x1,x2,x3,x4)

 k12=f1(t+h2,x1+h2*k11,x2+h2*k21,x3+h2*k31,x4+h2*k41)
 k22=f2(t+h2,x1+h2*k11,x2+h2*k21,x3+h2*k31,x4+h2*k41)
 k32=f3(t+h2,x1+h2*k11,x2+h2*k21,x3+h2*k31,x4+h2*k41)
 k42=f4(t+h2,x1+h2*k11,x2+h2*k21,x3+h2*k31,x4+h2*k41)

 k13=f1(t+h2,x1+h2*k12,x2+h2*k22,x3+h2*k32,x4+h2*k42)
 k23=f2(t+h2,x1+h2*k12,x2+h2*k22,x3+h2*k32,x4+h2*k42)
 k33=f3(t+h2,x1+h2*k12,x2+h2*k22,x3+h2*k32,x4+h2*k42)
 k43=f4(t+h2,x1+h2*k12,x2+h2*k22,x3+h2*k32,x4+h2*k42)

 k14=f1(t+h ,x1+h *k13,x2+h *k23,x3+h *k33,x4+h2*k43)
 k24=f2(t+h ,x1+h *k13,x2+h *k23,x3+h *k33,x4+h2*k43)
 k34=f3(t+h ,x1+h *k13,x2+h *k23,x3+h *k33,x4+h2*k43)
 k44=f4(t+h ,x1+h *k13,x2+h *k23,x3+h *k33,x4+h2*k43)

 t =t+h
 x1=x1+h6*(k11+2.0D0*(k12+k13)+k14)
 x2=x2+h6*(k21+2.0D0*(k22+k23)+k24)
 x3=x3+h6*(k31+2.0D0*(k32+k33)+k34)
```

```
 x4=x4+h6*(k41+2.0D0*(k42+k43)+k44)

end subroutine RKSTEP
```

## 5.2 Projectile Motion

Consider a particle in the constant gravitational field near the surface of the earth which moves with constant acceleration $\vec{g} = -g\,\hat{y}$ so that

$$\begin{array}{llll} x(t) & = x_0 + v_{0x}t & , \quad y(t) & = y_0 + v_{0y}t - \frac{1}{2}gt^2 \\ v_x(t) & = v_{0x} & , \quad v_y(t) & = v_{0y} - gt \\ a_x(t) & = 0 & , \quad a_y(t) & = -g \end{array} \tag{5.2}$$

The particle moves on a parabolic trajectory that depends on the initial conditions

$$\begin{aligned} (y - y_0) &= \left(\frac{v_{0y}}{v_{0x}}\right)(x - x_0) - \frac{1}{2}\frac{g}{v_{0x}^2}(x - x_0)^2 \\ &= \tan\theta\,(x - x_0) - \frac{\tan^2\theta}{4h_{\max}}(x - x_0)^2\,, \end{aligned} \tag{5.3}$$

where $\tan\theta = v_{0y}/v_{0x}$ is the direction of the initial velocity and $h_{\max}$ is the maximum height of the trajectory.

The acceleration $a_x(t) = 0$ $a_y(t) = -g$ ($a_x \leftrightarrow$ `f3` , $a_y \leftrightarrow$ `f4`) and the mechanical energy is coded in the file `rk2_g.f90`:

```
!=========================================================
!The acceleration functions f3,f4(t,x1,x2,v1,v2) provided
!by the user
!=========================================================
!Free fall in constant gravitational filed with
!g = -k2
real(8) function f3(t,x1,x2,v1,v2)
 implicit none
 real(8) :: t,x1,x2,v1,v2
 real(8) ::          k1,k2
 common  /couplings/k1,k2
 f3=0.0D0     !dx3/dt=dv1/dt=a1
end function f3
!---------------------------------------------------------
real(8) function f4(t,x1,x2,v1,v2)
 implicit none
 real(8) :: t,x1,x2,v1,v2
```

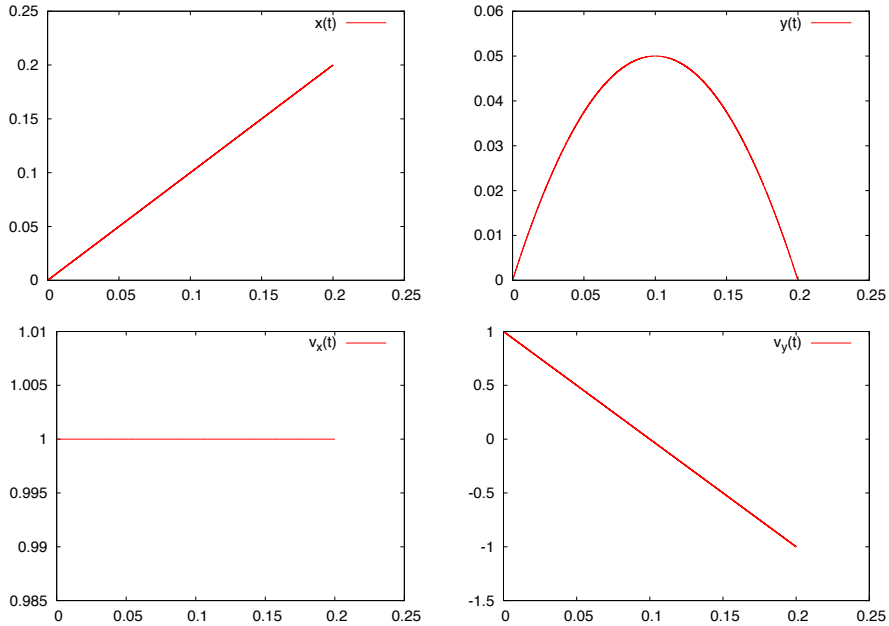



Figure 5.1: Plots of $x(t)$, $y(t)$, $v_x(t)$, $v_y(t)$ for a projectile fired in a constant gravitational field $\vec{g} = -10.0\,\hat{y}$ with initial velocity $\vec{v}_0 = \hat{x} + \hat{y}$.

```
 real(8) ::          k1,k2
 common  /couplings/k1,k2
 f4=-k1    !dx4/dt=dv2/dt=a2
end function f4
!-----------------------------------------------------------------
real(8) function energy(t,x1,x2,v1,v2)
 implicit none
 real(8) :: t,x1,x2,v1,v2
 real(8) ::          k1,k2
 common  /couplings/k1,k2
 energy = 0.5D0*(v1*v1+v2*v2) + k1*x2
end function energy
```

In order to calculate a projectile's trajectory you may use the following commands:

```
> gfortran -O2 rk2.f90 rk2_g.f90 -o rk2
> ./rk2
 Runge-Kutta Method for 4-ODEs Integration
 Enter coupling constants:
10.0 0.0
 k1=    10.000000        k2=    0.000000
```

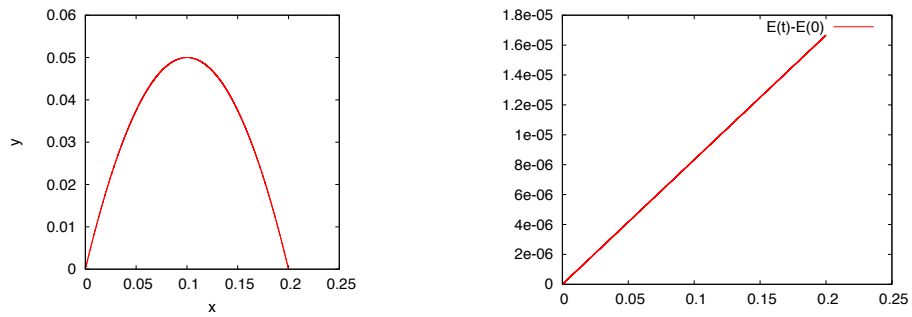



Figure 5.2: (Left) The parabolic trajectory of a projectile fired in a constant gravitational field $\vec{g} = -10.0\,\hat{y}$ with initial velocity $\vec{v}_0 = \hat{x} + \hat{y}$. (Right) The deviation of the projectile's energy from its initial value is due to numerical errors.

```
 Enter Nt,Ti,Tf,X10,X20,V10,V20:
20000 0.0 0.2 0.0 0.0 1.0 1.0
 Nt=        20000
 Time: Initial Ti =   0.000000     Final Tf=   0.200000
            X1(Ti)=   0.000000       X2(Ti)=   0.000000
            V1(Ti)=   1.000000       V2(Ti)=   1.000000
```

The analysis of the results contained in the file `rk2.dat` can be done using `gnuplot`:

```
gnuplot> set terminal x11 1
gnuplot> plot "rk2.dat" using 1:2 with lines title "x(t)"
gnuplot> set terminal x11 2
gnuplot> plot "rk2.dat" using 1:3 with lines title "y(t)"
gnuplot> set terminal x11 3
gnuplot> plot "rk2.dat" using 1:4 with lines title "vx(t)"
gnuplot> set terminal x11 4
gnuplot> plot "rk2.dat" using 1:5 with lines title "vy(t)"
gnuplot> set terminal x11 5
gnuplot> plot "rk2.dat" using 1:($6-1.0) w lines t "E(t)E-(0)"
gnuplot> set terminal x11 6
gnuplot> set size square
gnuplot> set title "Trajectory"
gnuplot> plot "rk2.dat" using 2:3 with lines notit
```

The results can be seen in figures 5.1 and 5.2. We note a small increase in the mechanical energy which is due to the accumulation of numerical errors.

We can animate the trajectory by writing a script of `gnuplot` com-

mands in a file rk2_animate.gpl

```
icount = icount+skip
plot  "<cat -n rk2.dat"  \
  using 3:($1<= icount ? $4: 1/0) with lines notitle
# pause 1
if(icount < nlines ) reread
```

Before calling the script, the user must set the values of the variables icount, skip and nlines. Each time gnuplot reads the script, it plots icount number of lines from rk2.dat. Then the script is read again and a new plot is made with skip lines more than the previous one, unless icount < nlines. The plotted "file" "<cat -n rk2.dat" is the standard output (stdout) of the command cat -n rk2.dat which prints to the stdout the contents of the file rk2.dat line by line, together with the line number. Therefore the plot command reads data which are the line number, the time, the coordinate $x$, the coordinate $y$ etc. The keyword using in

```
  using 3:($1<= icount ? $4: 1/0)
```

instructs the plot command to use the 3rd column on the horizontal axis and if the first column is less than icount ($1<= icount) put on the vertical axis the value of the 4th column if the first column is less than icount. Otherwise ($1 > icount) it prints an undefined number (1/0) which makes gnuplot print nothing at all. You may also uncomment the command pause if you want to make the animation slower. In order to run the script from gnuplot, issue the commands

```
gnuplot> icount = 10
gnuplot> skip   = 200
gnuplot> nlines = 20000
gnuplot> load "rk2_animate.gpl"
```

The scripts shown above can be found in the accompanying software. More scripts can be found there that automate many of the boring procedures. The usage of two of these is explained below. The first one is in the file rk2_animate.csh:

```
> ./rk2_animate.csh -h
Usage: rk2_animate.csh -t [sleep time] -d [skip points] <file>
Default file is rk2.dat
```

```
Other options:
   -x: set lower value in xrange
   -X: set lower value in xrange
   -y: set lower value in yrange
   -Y: set lower value in yrange
   -r: automatic determination of x-y range
> ./rk2_animate.csh -r -d 500 rk2.dat
```

The last line is a command that animates a trajectory read from the file `rk2.dat`. Each animation frame contains 500 more points than the previous one. The option `-r` calculates the plot range automatically. The option `-h` prints a short help message.

A more useful script is in the file `rk2.csh`.

```
> ./rk2.csh -h
Usage: rk2.csh -f <force> k1 k2 x10 x20 v10 v20 STEPS t0 tf
Other Options:
 -n Do not animate trajectory
Available forces (value of <force>):
1: ax=-k1                ay= -k2 y            Harmonic oscillator
2: ax= 0                 ay= -k1              Free fall
3: ax= -k2      vx       ay= -k2     vy - k1  Free fall + \
                                              air resistance ~ v
4: ax= -k2 |v| vx        ay= -k2 |v|vy - k1   Free fall + \
                                              air resistance ~ v^2
5: ax= k1*x1/r^3         ay= k1*x2/r^3        Coulomb Force
....
```

The option `-h` prints operating instructions. A menu of forces is available, and a choice can be made using the option `-f`. The rest of the command line consists of the parameters read by the program in `rk2.f90`, i.e. the coupling constants `k1, k2`, the initial conditions `x10, x20, v10, v20` and the integration parameters `STEPS, t0` and `tf`. For example, the commands

```
> rk2.csh -f 2 -- 10.0 0.0 0.0 0.0 1.0 1.0 20000 0.0 0.2
> rk2.csh -f 1 -- 16.0 1.0 0.0 1.0 1.0 0.0 20000 0.0 6.29
> rk2.csh -f 5 -- 10.0 0.0 -10 0.2 10. 0.0 20000 0.0 3.00
```

compute the trajectory of a particle in the constant gravitational field discussed above, the trajectory of an anisotropic harmonic oscillator (`k1` = $a_x = -\omega_1^2 x$, `k2` = $a_y = -\omega_2^2 y$) and the scattering of a particle in a Coulomb field – try them! I hope that you will have enough curiosity to look "under the hood" of the scripts and try to modify them or create new ones. Some advise to the lazy guys: If you need to program your

own force field follow the recipe: Write the code of your acceleration field in a file named e.g. `rk2_myforce.f90` as we did with `rk2_g.f90`. Edit the file `rk2.csh` and modify the line

```
set forcecode = (hoc g vg v2g cb)
```

to

```
set forcecode = (hoc g vg v2g cb myforce)
```

(the variable `$forcecode` may have more entries than the ones shown above). Count the order of the string `myforce`, which is 6 in our case. In order to access this force field from the command line, use the option `-f 6`:

```
> rk2.csh -f 6 -- .......
```

Now, we will study the effect of the air resistance on the motion of the projectile. For small velocities this is a force proportional to the velocity $\vec{F}_r = -mk\vec{v}$, therefore

$$\begin{aligned} a_x &= -kv_x \\ a_y &= -kv_y - g\,. \end{aligned} \tag{5.4}$$

By taking

$$\begin{aligned} x(t) &= x_0 + \frac{v_{0x}}{k}\left(1 - \mathrm{e}^{-kt}\right) \\ y(t) &= y_0 + \frac{1}{k}\left(v_{0y} + \frac{g}{k}\right)\left(1 - \mathrm{e}^{-kt}\right) - \frac{g}{k}t \\ v_x(t) &= v_{0x}\mathrm{e}^{-kt} \\ v_y(t) &= \left(v_{0y} + \frac{g}{k}\right)\mathrm{e}^{-kt} - \frac{g}{k}\,, \end{aligned} \tag{5.5}$$

we obtain the motion of a particle with terminal velocity $v_y(+\infty) = -g/k$ ($x(+\infty) = \text{const.}$, $y(+\infty) \sim t$).

The acceleration caused by the air resistance is programmed in the file (`k1` $\leftrightarrow g$, `k2` $\leftrightarrow k$ ) `rk2_vg.f90`:

```
!=========================================================
!The acceleration functions f3,f4(t,x1,x2,v1,v2) provided
!by the user
```

```
!=========================================================
!Free fall in constant gravitational filed with
!ax = -k2 vx     ay = -k2 vy - k1
real(8) function f3(t,x1,x2,v1,v2)
 implicit none
 real(8) :: t,x1,x2,v1,v2
 real(8) ::          k1,k2
 common  /couplings/k1,k2
 f3=-k2*v1    !dx3/dt=dv1/dt=a1
end function f3
!---------------------------------------------------------
real(8) function f4(t,x1,x2,v1,v2)
 implicit none
 real(8) :: t,x1,x2,v1,v2
 real(8) ::          k1,k2
 common  /couplings/k1,k2
 f4=-k2*v2-k1    !dx4/dt=dv2/dt=a2
end function f4
```

The results are shown in figure 5.3 where we see the effect of an increasing air resistance on the particle trajectory. The effect of a resistance force of the form $\vec{F}_r = -mkv^2\hat{v}$ is shown in figure 5.4.

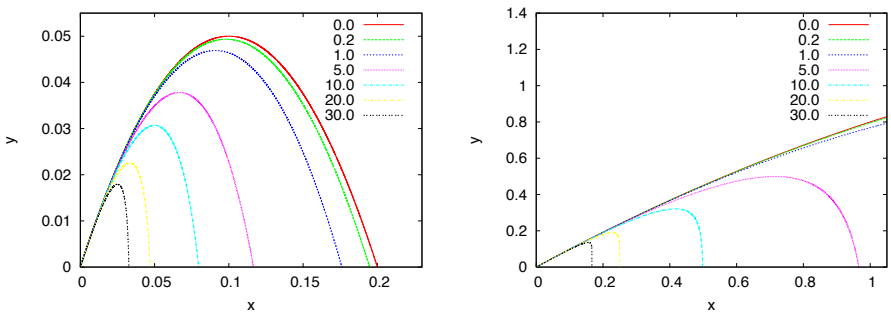



Figure 5.3: The trajectory of a projectile moving in a constant gravitational field $\vec{g} = -10\,\hat{y}$ with air resistance causing acceleration $\vec{a}_r = -k\vec{v}$ for $k = 0, 0.2, 1, 5, 10, 20, 30$. The left plot has $\vec{v}(0) = \hat{x} + \hat{y}$ and the right plot has $\vec{v}(0) = 5\hat{x} + 5\hat{y}$.

## 5.3 Planetary Motion

Consider the simple planetary model of a "sun" of mass $M$ and a planet "earth" at distance $r$ from the sun and mass $m$ such that $m \ll M$. Ac-

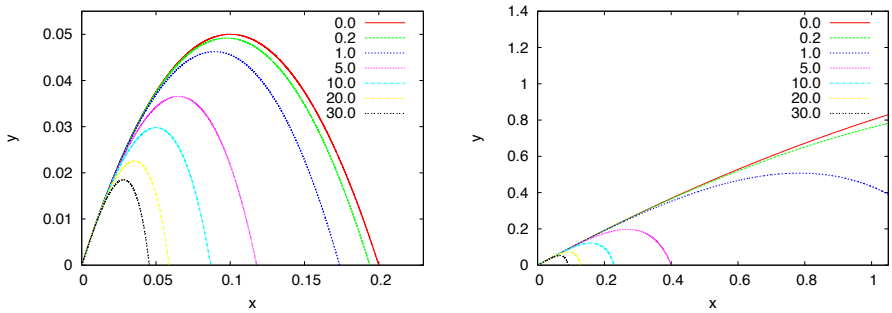



Figure 5.4: The trajectory of a projectile moving in a constant gravitational field $\vec{g} = -10\,\hat{y}$ with air resistance causing acceleration $\vec{a}_r = -kv^2\hat{v}$ for $k = 0, 0.2, 1, 5, 10, 20, 30$. The left plot has $\vec{v}(0) = \hat{x} + \hat{y}$ and the right plot has $\vec{v}(0) = 5\hat{x} + 5\hat{y}$.

cording to Newton's law of gravity, the earth's acceleration is

$$\vec{a} = \vec{g} = -\frac{GM}{r^2}\hat{r} = -\frac{GM}{r^3}\vec{r}, \tag{5.6}$$

where $G = 6.67 \times 10^{-11}\frac{\text{m}^3}{\text{kgr}\cdot\text{sec}^2}$, $M = 1.99 \times 10^{30}$kgr, $m = 5.99 \times 10^{24}$kgr. When the hypothesis $m \ll M$ is not valid, the two body problem is reduced to that of the one body problem with the mass replaced by the reduced mass $\mu$

$$\frac{1}{\mu} = \frac{1}{m} + \frac{1}{M}\,.$$

The force of gravity is a central force. This implies conservation of the angular momentum $\vec{L} = \vec{r} \times \vec{p}$ with respect to the center of the force, which in turn implies that the motion is confined on one plane. We choose the $z$ axis so that

$$\vec{L} = L_z\hat{k} = m(xv_y - yv_x)\hat{k}\,. \tag{5.7}$$

The force of gravity is conservative and the mechanical energy

$$E = \frac{1}{2}mv^2 - \frac{GmM}{r} \tag{5.8}$$

is conserved. If we choose the origin of the coordinate axes to be the center of the force, the equations of motion (5.6) become

$$\begin{aligned} a_x &= -\frac{GM}{r^3}\,x \\ a_y &= -\frac{GM}{r^3}\,y\,, \end{aligned} \tag{5.9}$$

where $r^2 = x^2 + y^2$. This is a system of two coupled differential equations for the functions $x(t)$, $y(t)$. The trajectories are conic sections which are either an ellipse (bound states - "planet"), a parabola (e.g. escape to infinity when the particle starts moving with speed equal to the escape velocity) or a hyperbola (e.g. scattering).

Kepler's third law of planetary motion states that the orbital period $T$ of a planet satisfies the equation

$$T^2 = \frac{4\pi^2}{GM} a^3 \,, \tag{5.10}$$

where $a$ is the semi-major axis of the elliptical trajectory. The eccentricity is a measure of the deviation of the trajectory from being circular

$$e = \sqrt{1 - \frac{b^2}{a^2}} \,, \tag{5.11}$$

where $b$ is the semi-minor axis. The eccentricity is 0 for the circle and tends to 1 as the ellipse becomes more and more elongated. The foci $F_1$ and $F_2$ are located at a distance $ea$ from the center of the ellipse. They have the property that for every point on the ellipse

$$PF_1 + PF_2 = 2a \,. \tag{5.12}$$

The acceleration given to the particle by Newton's force of gravity is programmed in the file `rk2_cb.f90`:

```
!=========================================================
!The acceleration functions f3,f4(t,x1,x2,v1,v2) provided
!by the user
!=========================================================
!Motion in Coulombic potential:
!ax= k1*x1/r^3 ay= k1*x2/r^3
real(8) function f3(t,x1,x2,v1,v2)
 implicit none
 real(8) :: t,x1,x2,v1,v2
 real(8) ::          k1,k2
 common  /couplings/k1,k2
 real(8) :: r2,r3
 r2=x1*x1+x2*x2
 r3=r2*sqrt(r2)
 if(r3.gt.0.0D0)then
  f3=k1*x1/r3                !dx3/dt=dv1/dt=a1
 else
  f3=0.0D0
```

```
 endif
end function f3
!-----------------------------------------------------------
real(8) function f4(t,x1,x2,v1,v2)
 implicit none
 real(8) :: t,x1,x2,v1,v2
 real(8) ::          k1,k2
 common  /couplings/k1,k2
 real(8) :: r2,r3
 r2=x1*x1+x2*x2
 r3=r2*sqrt(r2)
 if(r3.gt.0.0D0)then
  f4=k1*x2/r3                !dx4/dt=dv2/dt=a2
 else
  f4=0.0D0
 endif
end function f4
!-----------------------------------------------------------
real(8) function energy(t,x1,x2,v1,v2)
 implicit none
 real(8) :: t,x1,x2,v1,v2
 real(8) ::          k1,k2
 common  /couplings/k1,k2
 real(8) :: r
 r=sqrt(x1*x1+x2*x2)
 if( r .gt. 0.0D0)then
  energy = 0.5D0*(v1*v1+v2*v2) + k1/r
 else
  energy = 0.0D0
 endif
end function energy
```

We set `k1=` $-GM$ and take special care to avoid hitting the center of the force, the singular point at $(0,0)$. The same code can be used for the electrostatic Coulomb field with `k1=` $qQ/4\pi\epsilon_0 m$.

At first we study trajectories which are bounded. We set $GM = 10$, $x(0) = 1.0$, $y(0) = 0$, $v_{0x} = 0$ and vary $v_{0y}$. We measure the period $T$ and the length of the semi axes of the resulting ellipse. The results can be found in table 5.1. Some of the trajectories are shown in figure 5.5. There we can see the dependence of the size of the ellipse on the period. Figure 5.6 confirms Kepler's third law of planetary motion given by equation (5.10).

In order to confirm Kepler's third law of planetary motion numerically, we take the logarithm of both sides of equation (5.10)

$$\ln T = \frac{3}{2}\ln a + \frac{1}{2}\ln\left(\frac{4\pi^2}{GM}\right) . \tag{5.13}$$

| $v_{0x}$ | $T/2$ | $2a$ |
|---|---|---|
| 3.2 | 1.032 | 2.051 |
| 3.4 | 1.282 | 2.371 |
| 3.6 | 1.682 | 2.841 |
| 3.8 | 2.398 | 3.598 |
| 4.0 | 3.9288 | 5.002 |
| 4.1 | 5.5164 | 6.272 |
| 4.2 | 8.6952 | 8.481 |
| 4.3 | 16.95 | 13.256 |
| 4.35 | 28.168 | 18.6 |
| 4.38 | 42.81 | 24.58 |
| 4.40 | 61.8 | 31.393 |
| 4.42 | 99.91 | 43.252 |

Table 5.1: The results for the period $T$ and the length of the semi-major axis $a$ of the trajectory of planetary motion for $GM = 10$, $x(0) = 1.0$, $y(0) = 0$, $v_{0y} = 0$.

Therefore, the points $(\ln a, \ln T)$ lie on a straight line. Using a linear least squares fit we calculate the slope and the intercept which should be equal to $\frac{3}{2}$ and $1/2 \ln (4\pi^2/GM)$ respectively. This is left as an exercise.

In the case where the initial velocity of the particle becomes larger than the escape velocity $v_e$, the particle escapes from the influence of the gravitational field to infinity. The escape velocity corresponds to zero mechanical energy, which gives

$$v_e^2 = \frac{2GM}{r} \,. \tag{5.14}$$

When $GM = 10$, $x(0) = 1.0$, $y(0) = 0$, we obtain $v_e \approx 4.4721\ldots$. The numerical calculation of $v_e$ is left as an exercise.

## 5.4 Scattering

In this section we consider scattering of particles from a central potential[1]. We assume particles that follow unbounded trajectories that start from infinity and move almost free from the influence of the force field towards its center. When they approach the region of interaction they get deflected and get off to infinity in a new direction. We say that the particles have been scattered and that the angle between their original and final direction

[1] We refer the reader to [35], chapter 4.

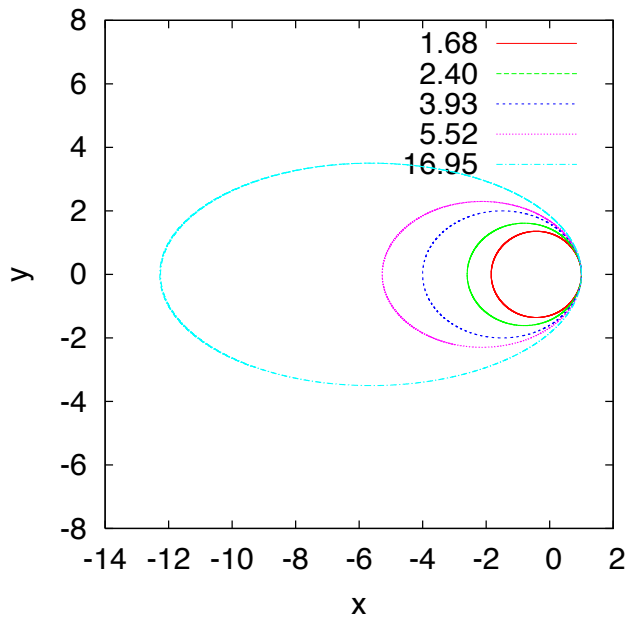



Figure 5.5: Planetary trajectories for $GM = 10$, $x(0) = 1.0$, $y(0) = 0$, $v_{0y} = 0$ and $v_{0x} = 3.6, 3.8, 4.0, 4.1, 4.3$. The numbers are the corresponding half periods.

is the scattering angle $\theta$. Scattering problems are interesting because we can infer to the properties of the scattering potential from the distribution of the scattering angle. This approach is heavily used in today's particle accelerators for the study of fundamental interactions between elementary particles.

First we will discuss scattering of small hard spheres of radius $r_1$ by other hard spheres or radius $R_2$. The interaction potential[2] is given by

$$V(r) = \begin{cases} 0 & r > R_2 + r_1 \\ \infty & r < R_2 + r_1 \end{cases} , \tag{5.15}$$

where $r$ is the distance between the center of $r_1$ from the center of $R_2$. Assume that the particles in the beam do not interact with each other and that there is only one collision per scattering. Let $J$ be the intensity of the beam[3] and $A$ its cross sectional area. Assume that the target has $n$ particles per unit area. The cross sectional area of the interaction is $\sigma = \pi(r_1 + R_2)^2$ where $r_1$ and $R_2$ are the radii of the scattered particles

[2]The so called hard core potential.

[3]The number of particles crossing a surface perpendicular to the beam per unit time and unit area.

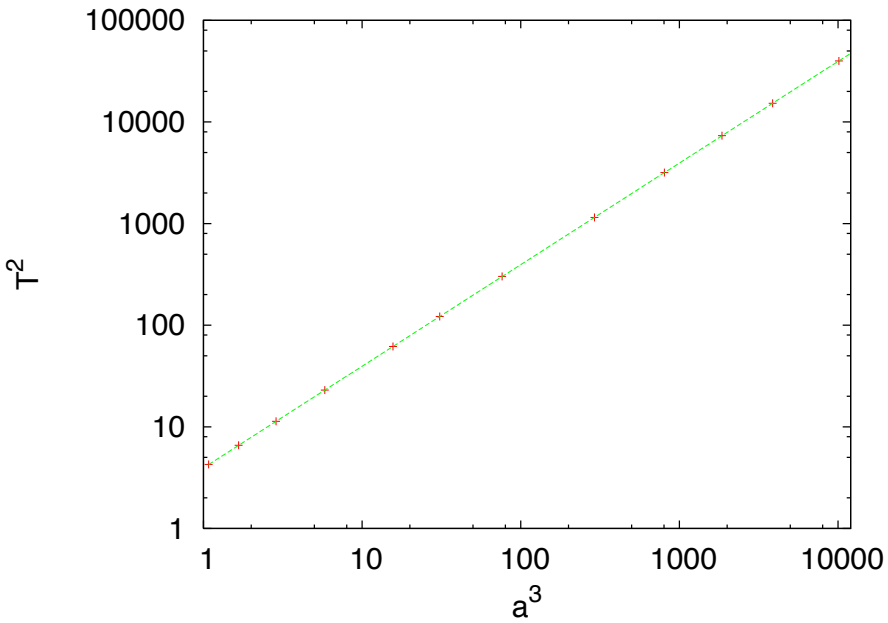



Figure 5.6: Kepler's third law of planetary motion for $GM = 10$. The points are the measurements taken from table 5.1. The solid line is the known analytic solution (5.10).

and targets respectively (see figure (5.8)): All the spheres of the beam which lie outside this area are not scattered by the particular target. The total interaction cross section is

$$\Sigma = nA\sigma, \tag{5.16}$$

where $nA$ is the total number of target spheres which lie within the beam. On the average, the scattering rate is

$$N = J\Sigma = JnA\sigma\,. \tag{5.17}$$

The above equation is the definition of the total scattering cross section $\sigma$ of the interaction. The differential cross section $\sigma(\theta)$ is defined by the relation

$$dN = JnA\sigma(\theta)\,d\Omega\,, \tag{5.18}$$

where $dN$ is the number of particles per unit time scattered within the solid angle $d\Omega$. The total cross section is

$$\sigma_{tot} = \int_\Omega \sigma(\theta)\,d\Omega = \int \sigma(\theta)\sin\theta\,d\theta d\phi = 2\pi \int \sigma(\theta)\sin\theta\,d\theta\,. \tag{5.19}$$

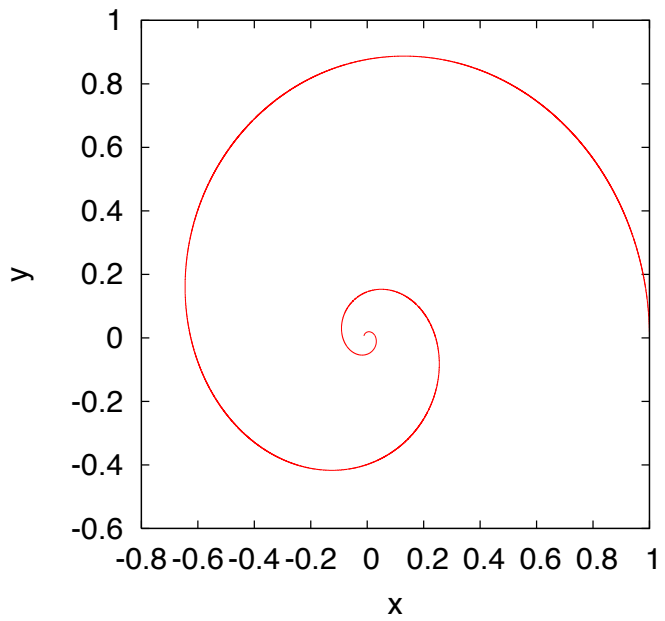



Figure 5.7: The spiral orbit of a particle moving under the influence of a central force $\vec{F} = -k/r^3\hat{r}$.

In the last relation we used the cylindrical symmetry of the interaction with respect to the axis of the collision. Therefore

$$\sigma(\theta) = \frac{1}{nAJ}\frac{dN}{2\pi \sin\theta\, d\theta}\,. \tag{5.20}$$

This relation can be used in experiments for the measurement of the differential cross section by measuring the rate of detection of particles within the space contained in between two cones defined by the angles $\theta$ and $\theta + d\theta$. This is the relation that we will use in the numerical calculation of $\sigma(\theta)$.

Generally, in order to calculate the differential cross section we shoot a particle at a target as shown in figure 5.9. The scattering angle $\theta$ depends on the impact parameter $b$. The part of the beam crossing the ring of radius $b(\theta)$, thickness $db$ and area $2\pi b\, db$ is scattered in angles between $\theta$ and $\theta + d\theta$. Since there is only one particle at the target we have that $nA = 1$. The number of particles per unit time crossing the ring is $J2\pi b\, db$, therefore

$$2\pi b(\theta)\, db = -2\pi\sigma(\theta)\sin\theta\, d\theta \tag{5.21}$$

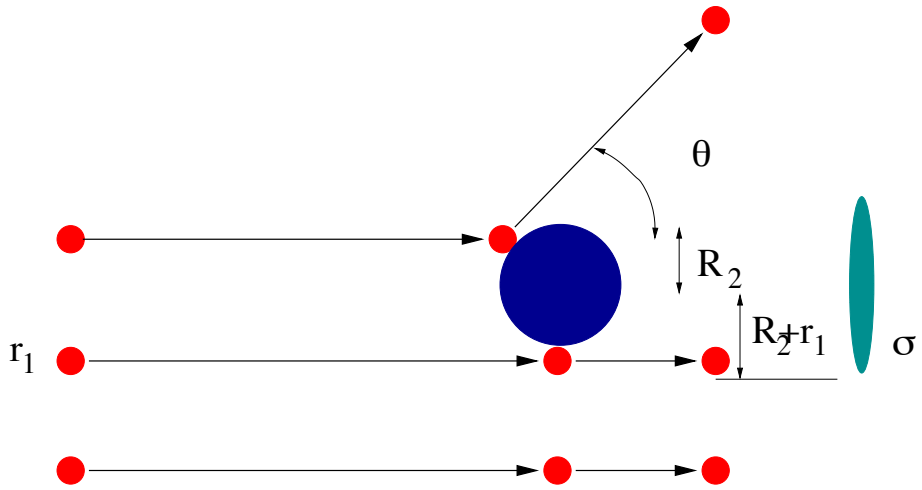



Figure 5.8: Scattering of hard spheres. The scattering angle is $\theta$. The cross sectional area $\sigma$ is shown to the right.

(the $-$ sign is because as $b$ increases, $\theta$ decreases). From the potential we can calculate $b(\theta)$ and from $b(\theta)$ we can calculate $\sigma(\theta)$. Conversely, if we measure $\sigma(\theta)$, we can calculate $b(\theta)$.

### 5.4.1 Rutherford Scattering

The scattering of a charged particle with charge $q$ ("electron") in a Coulomb potential of a much heavier charge $Q$ ("nucleus") is called Rutherford scattering. In this case, the interaction potential is given by

$$V(r) = \frac{1}{4\pi\epsilon_0}\frac{Q}{r}\,, \tag{5.22}$$

which accelerates the particle with acceleration

$$\vec{a} = \frac{qQ}{4\pi\epsilon_0 m}\frac{\hat{r}}{r^2} \equiv \alpha\frac{\vec{r}}{r^3}\,. \tag{5.23}$$

The energy of the particle is $E = \frac{1}{2}mv^2$ and the magnitude of its angular momentum is $l = mvb$, where $v \equiv |\vec{v}|$. The dependence of the impact parameter on the scattering angle is [35]

$$b(\theta) = \frac{\alpha}{v^2}\cot\frac{\theta}{2}\,. \tag{5.24}$$

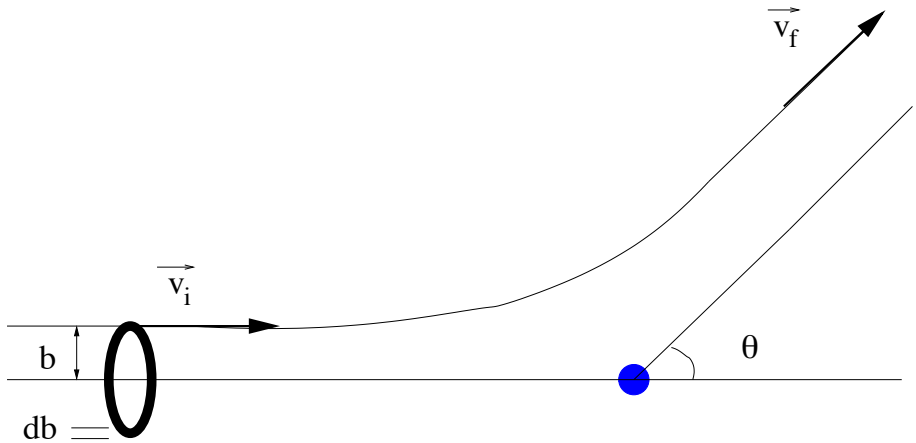



Figure 5.9: Beam particles passing through the ring $2\pi b db$ are scattered within the solid angle $d\Omega = 2\pi sin\theta\, d\theta$.

Using equation (5.21) we obtain

$$\sigma(\theta) = \frac{\alpha^2}{4}\frac{1}{v^4}\sin^{-4}\frac{\theta}{2}\,. \tag{5.25}$$

Consider the scattering trajectories. The results for same charges are shown in figure 5.10. A similar figure is obtained in the case of opposite charges. In the latter case we have to take special care for small impact parameters $b < 0.2$ where the scattering angle is $\approx 1$. A large number of integration steps is needed in order to obtain the desired accuracy. A useful monitor of the accuracy of the calculation is the measurement of the energy of the particle which should be conserved. The results are shown in table 5.2. We will now describe a method for calculating the cross section by using equation (5.20). Alternatively we could have used equation (5.21) and perform a numerical calculation of the derivatives. This is left as an exercise for the reader. Our calculation is more like an experiment. We place a "detector" that "detects" particles scattered within angles $\theta$ and $\theta+\delta\theta$. For this reason we split the interval $[0,\pi]$ in $N_b$ bins so that $\delta\theta = \pi/N_b$. We perform "scattering experiments" by varying $b \in [b_m, b_M]$ with step $\delta b$. Due to the symmetry of the problem we fix $\phi$ to be a constant, therefore a given $\theta$ corresponds to a cone with an opening angle $\theta$ and an apex at the center of scattering. For given $b$ we measure the scattering angle $\theta$ and record the number of particles per unit time $\delta N \propto b\delta b$. The latter is proportional to the area of the ring of radius $b$. All we need now is the beam intensity $J$ which is the total number of particles per unit time $J \propto \sum_i b\delta b$ (note than in the ratio $\delta N/J$ the

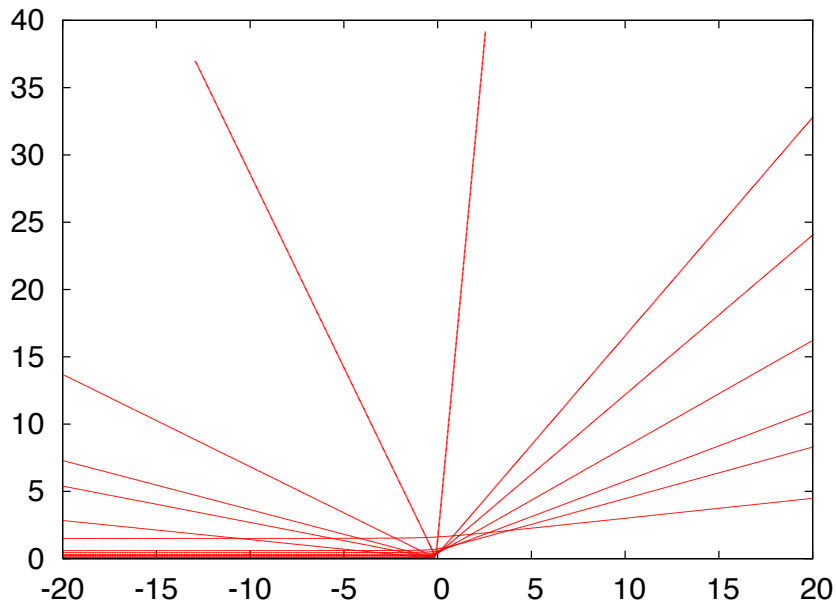



Figure 5.10: Rutherford scattering trajectories. We set `k1` $\equiv \frac{qQ}{4\pi\epsilon_0 m} = 1$ (see code in the file `rk2_cb.f90`) and $b = 0.08$, $0.015$, $0.020$, $0.035$, $0.080$, $0.120$, $0.200$, $0.240$, $0.320$, $0.450$, $0.600$, $1.500$. The initial position of the particle is at $x(0) = -50$ and its initial velocity is $v = 3$ in the $x$ direction. The number of integration steps is 1000, the initial time is 0 and the final time is 30.

proportionality constant and $\delta b$ cancel) and the solid angle $2\pi \sin(\theta)\, \delta\theta$. Finally we can easily use equation (5.19) in order to calculate the total cross section $\sigma_{tot}$. The program that performs this calculation is in the file `scatter.f90` and it is a simple modification of the program in `rk2.f90`:

```
!=========================================================
!Program that computes scattering cross-section of a central
!force on the plane. The user should first check that the
!parameters used, lead to a free state in the end.
!  **  X20 is the impact parameter b  **
!A 4 ODE system is solved using Runge-Kutta Method
!User must supply derivatives
!dx1/dt=f1(t,x1,x2,x3,x4) dx2/dt=f2(t,x1,x2,x3,x4)
!dx3/dt=f3(t,x1,x2,x3,x4) dx4/dt=f4(t,x1,x2,x3,x4)
!as real(8) functions
!Output is written in file scatter.dat
!=========================================================
program scatter_cross_section
 implicit none
```

| $b$ | $\theta_n$ | $\theta_a$ | $\Delta E/E$ | Nt |
|---|---|---|---|---|
| 0.008 | 2.9982 | 2.9978 | $1.06\,10^{-5}$ | 5000 |
| 0.020 | 2.7861 | 2.7854 | $6.25\,10^{-5}$ | 5000 |
| 0.030 | 2.6152 | 2.6142 | $1.29\,10^{-4}$ | 5000 |
| 0.043 | 2.4043 | 2.4031 | $2.31\,10^{-4}$ | 5000 |
| 0.056 | 2.2092 | 2.2079 | $3.27\,10^{-4}$ | 5000 |
| 0.070 | 2.0184 | 2.0172 | $4.07\,10^{-4}$ | 5000 |
| 0.089 | 1.7918 | 1.7909 | $4.70\,10^{-4}$ | 5000 |
| 0.110 | 1.5814 | 1.5808 | $4.82\,10^{-4}$ | 5000 |
| 0.130 | 1.4147 | 1.4144 | $4.59\,10^{-4}$ | 5000 |
| 0.160 | 1.2138 | 1.2140 | $3.97\,10^{-4}$ | 5000 |
| 0.200 | 1.0137 | 1.0142 | $3.08\,10^{-4}$ | 5000 |
| 0.260 | 0.8070 | 0.8077 | $2.04\,10^{-4}$ | 5000 |
| 0.360 | 0.5979 | 0.5987 | $1.07\,10^{-4}$ | 5000 |
| 0.560 | 0.3910 | 0.3917 | $3.83\,10^{-5}$ | 5000 |
| 1.160 | 0.1905 | 0.1910 | $5.58\,10^{-6}$ | 5000 |

Table 5.2: Scattering angles of Rutherford scattering. We set k1 $\equiv \frac{qQ}{4\pi\epsilon_0 m} = 1$ (see file rk2_cb.f90) and study the resulting trajectories for the values of $b$ shown in column 1. $\theta_n$ is the numerically calculated scattering angle and $\theta_a$ is the one calculated from equation (5.24). The ratio $\Delta E/E$ shows the change in the particle's energy due to numerical errors. The last column is the number of integration steps. The particle's initial position is at $x(0) = -50$ and initial velocity $\vec{v} = 3\hat{x}$.

```
 integer,parameter    :: P=1010000
 real(8),dimension(P):: T,X1,X2,V1,V2
 real(8) :: Ti,Tf,X10,X20,V10,V20
 real(8) :: X20F,dX20 !max impact parameter and step
 integer :: Nt
 integer :: i
 real(8) ::         k1,k2
 common /couplings/k1,k2
 integer, parameter :: Nbins=20
 integer :: index
 real(8) :: angle,bins(Nbins),Npart
 real(8),parameter :: PI      =3.14159265358979324D0
 real(8),parameter :: rad2deg=180.0D0/PI
 real(8),parameter :: dangle =PI/Nbins
 real(8) R,density,dOmega,sigma,sigmatot
!Input:
 print *,'Runge-Kutta Method for 4-ODEs Integration'
 print *,'Enter coupling constants:'
 read  *, k1,k2
 print *,'k1= ',k1,' k2= ',k2
```

| $b$ | $\theta_n$ | $\theta_a$ | $\Delta E/E$ | STEPS |
|---|---|---|---|---|
| 0.020 | 0.577 | 2.785 | 0.47 | 150000 |
| 0.030 | 2.466 | 2.614 | 0.22 | 150000 |
| 0.043 | 2.333 | 2.403 | $8.62\,10^{-2}$ | 150000 |
| 0.056 | 2.180 | 2.208 | $2.88\,10^{-2}$ | 150000 |
| 0.070 | 2.006 | 2.017 | $1.07\,10^{-2}$ | 150000 |
| 0.089 | 1.779 | 1.791 | $8.94\,10^{-3}$ | 60000 |
| 0.110 | 1.570 | 1.581 | $7.07\,10^{-3}$ | 30000 |
| 0.130 | 1.406 | 1.414 | $5.25\,10^{-3}$ | 20000 |
| 0.160 | 1.198 | 1.214 | $8.63\,10^{-3}$ | 5000 |
| 0.200 | 1.007 | 1.014 | $3.71\,10^{-3}$ | 5000 |
| 0.260 | 0.8057 | 0.8077 | $1.40\,10^{-3}$ | 5000 |
| 0.360 | 0.5988 | 0.5987 | $4.32\,10^{-4}$ | 5000 |
| 0.560 | 0.3923 | 0.3917 | $9.40\,10^{-5}$ | 5000 |
| 1.160 | 0.1913 | 0.1910 | $8.61\,10^{-6}$ | 5000 |

Table 5.3: Rutherford scattering of opposite charges with $\frac{qQ}{4\pi\epsilon_0 m} = -1$. The table is similar to table 5.2. We observe the numerical difficulty for small impact parameters.

```
 print *,'Enter Nt,Ti,Tf,X10,X20,V10,V20:'
 read  *,        Nt,Ti,TF,X10,X20,V10,V20
 print *,'Enter final impact parameter X20F and step dX20:'
 read  *, X20F,dX20
 print *,'Nt = ',Nt
 print *,'Time: Initial Ti =',Ti,   ' Final Tf=',Tf
 print *,'            X1(Ti)=',X10, ' X2(Ti)=',X20
 print *,'            V1(Ti)=',V10, ' V2(Ti)=',V20
 print *,'Impact par X20F  =',X20F,' dX20  =',dX20

 open(unit=11,file='scatter.dat')
 bins   = 0.0d0
!The Calculation:
 Npart  = 0.0D0
 X20    = X20 + dX20/2.0D0 !starts in middle of first interval
 do while (X20 .lt. X20F )
  call RK(T,X1,X2,V1,V2,Ti,Tf,X10,X20,V10,V20,Nt)
! Take absolute value due to symmetry:
  angle = DABS(atan2(V2(Nt),V1(Nt)))
!Output: The final angle. Check if almost constant
  write(11,*) '@ ', X20, angle,&
       DABS(atan2(V2(Nt-50),V1(Nt-50))),&
       k1/V10**2/tan(angle/2.0D0)
!Update histogram:
  index       = int(angle/dangle)+1
```

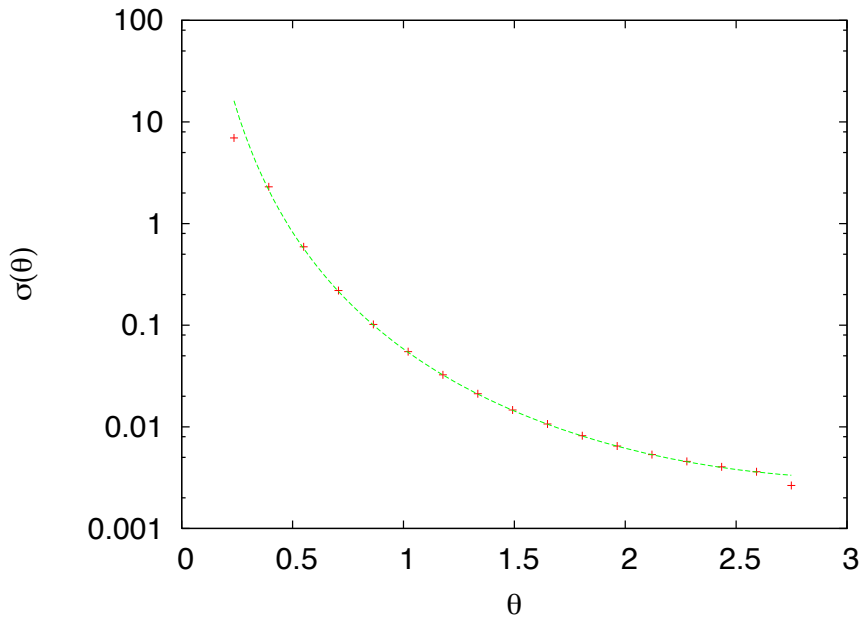



Figure 5.11: Differential cross section of the Rutherford scattering. The solid line is the function (5.25) for $\alpha = 1$, $v = 3$. We set $\frac{qQ}{4\pi\epsilon_0 m} = 1$. The particle's initial position is $x(0) = -50$ and its initial velocity is $\vec{v} = 3\hat{x}$. We used 5000 integration steps, initial time equal to 0 and final time equal to 30. The impact parameter varies between 0.02 and 1 with step equal to 0.0002.

```
!Number of incoming particles per unit time
!is proportional to radius of ring
!of radius X20, the impact parameter:
  !db is cancelled from density
  bins(index) = bins(index) +  X20
  Npart       = Npart       +  X20 !<-- i.e. from here
  X20         = X20         + dX20
 enddo
!Print scattering cross section:
 R         = X20              !beam radius
 density   = Npart/(PI*R*R)   !beam flux density J
 sigmatot  = 0.0D0            !total cross section
 do i=1,Nbins
  angle    = (i-0.5D0)*dangle
  dOmega   = 2.0D0*PI*sin(angle)*dangle !d(Solid Angle)
  sigma    = bins(i)/(density*dOmega)
  if(sigma.gt.0.0D0) write(11,*) 'ds= ',&
                     angle,angle*rad2deg,sigma
  sigmatot = sigmatot + sigma*dOmega
 enddo
```

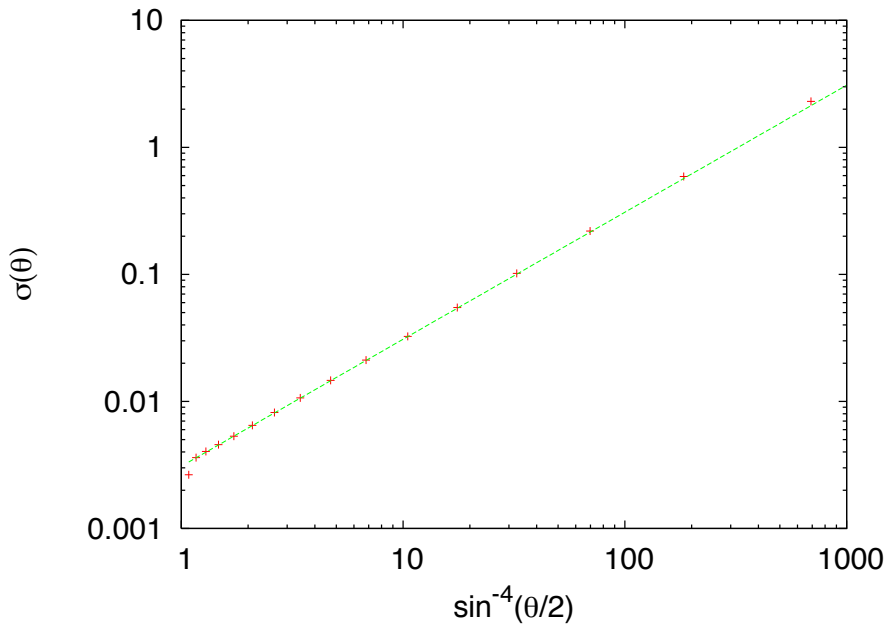



Figure 5.12: Differential cross section of the Rutherford scattering like in figure 5.11. The solid line is the function $1/(4\times 3^4)x$ from which we can deduce the functional form of $\sigma(\theta)$.

```
  write(11,*) 'sigmatot= ',sigmatot
  close(11)
end program scatter_cross_section
```

The results are recorded in the file `scatter.dat`. An example session that reproduces figures 5.11 and 5.12 is

```
> gfortran scatter.f90 rk2_cb.f90 -o scatter
> ./scatter
 Runge-Kutta Method for 4-ODEs Integration
 Enter coupling constants:
1.0 0.0
 k1=    1.00000        k2=    0.00000
 Enter Nt,Ti,Tf,X10,X20,V10,V20:
5000 0 30 -50 0.02 3 0
 Enter final impact parameter X20F and step dX20:
1 0.0002
 Nt=          5000
 Time: Initial T0 =    0.00000  Final TF= 30.00000
            X1(T0)= -50.00000    X2(T0)=  2.00000E-002
            V1(T0)=   3.00000    V2(T0)=  0.00000
```

```
 Impact par X20F  =   1.00000    dX20  =  2.00000E-004
```

The results can be plotted with the `gnuplot` commands:

```
gnuplot> set log
gnuplot> plot [:1000] "<grep ds= scatter.dat" \
 u ((sin($2/2))**(-4)):($4) notit,\
          (1./(4.*3.**4))*x notit
gnuplot> unset log
gnuplot> set log y
gnuplot> plot [:] "<grep ds= scatter.dat" u 2:4 notit, \
 (1./(4.*3.**4))*(sin(x/2))**(-4) notit
```

The results are in a very good agreement with the theoretical ones given by (5.25). The next step will be to study other central potentials whose solution is not known analytically.

### 5.4.2 More Scattering Potentials

Consider scattering from a force field

$$\vec{F} = f(r)\,\hat{r}\,, \qquad f(r) = \begin{cases} \frac{1}{r^2} - \frac{r}{a^3} & r \le a \\ 0 & r > a \end{cases} . \tag{5.26}$$

This is a very simple classical model of the scattering of a positron $e^+$ by the hydrogen atom. The positron has positive charge $+e$ and the hydrogen atom consists of a positively charged proton with charge $+e$ in an electron cloud of opposite charge $-e$. We set the scales so that $m_{e^+} = 1$ and $e^2/4\pi\epsilon_0 = 1$. We will perform a numerical calculation of $b(\theta)$, $\sigma(\theta)$ and $\sigma_{tot}$.

The potential energy is given by

$$f(r) = -\frac{dV(r)}{dr} \Rightarrow V(r) = \frac{1}{r} + \frac{r^2}{2a^2} - \frac{3}{2a}\,. \tag{5.27}$$

where $V(r) = 0$ for $r \ge a$. The program containing the calculation of the acceleration caused by this force can be found in the file `rk_hy.f90`:

```
!=========================================================
!The acceleration functions f3,f4(t,x1,x2,v1,v2) provided
!by the user
!=========================================================
!Motion in hydrogen atom + positron:
!f(r) = 1/r^2-r/k1^3
```

```
!ax= f(r)*x1/r ay= f(r)*x2/r
real(8) function f3(t,x1,x2,v1,v2)
 implicit none
 real(8) :: t,x1,x2,v1,v2
 real(8) ::          k1,k2
 common  /couplings/k1,k2
 real(8) :: r2,r,fr
 r2=x1*x1+x2*x2
 r =sqrt(r2)
 if(r .le.k1    .and. r2.gt.0.0D0)then
  fr = 1/r2-r/k1**3
 else
  fr = 0.0D0
 endif

 if(fr.gt.0.0D0 .and. r .gt.0.0D0)then
  f3=fr*x1/r              !dx3/dt=dv1/dt=a1
 else
  f3=0.0D0
 endif
end function f3
!-----------------------------------------------------------
real(8) function f4(t,x1,x2,v1,v2)
 implicit none
 real(8) :: t,x1,x2,v1,v2
 real(8) ::          k1,k2
 common  /couplings/k1,k2
 real(8) :: r2,r,fr
 r2=x1*x1+x2*x2
 r =sqrt(r2)
 if(r .le.k1    .and. r2.gt.0.0D0)then
  fr = 1/r2-r/k1**3
 else
  fr = 0.0D0
 endif

 if(fr.gt.0.0D0 .and. r .gt.0.0D0)then
  f4=fr*x2/r              !dx3/dt=dv1/dt=a1
 else
  f4=0.0D0
 endif
end function f4
!-----------------------------------------------------------
real(8) function energy(t,x1,x2,v1,v2)
 implicit none
 real(8) :: t,x1,x2,v1,v2
 real(8) ::          k1,k2
 common  /couplings/k1,k2
 real(8) :: r,Vr
```

```
 r=sqrt(x1*x1+x2*x2)
 if( r .le.k1   .and. r .gt.0.0D0)then
  Vr = 1/r + 0.5D0*r*r/k1**3 - 1.5D0 / k1
 else
  Vr = 0.0D0
 endif
 energy  = 0.5D0*(v1*v1+v2*v2) + Vr
end function energy
```

The results are shown in figures 5.13–5.14. We find that $\sigma_{tot} = \pi a^2$ (see problem 5.10).

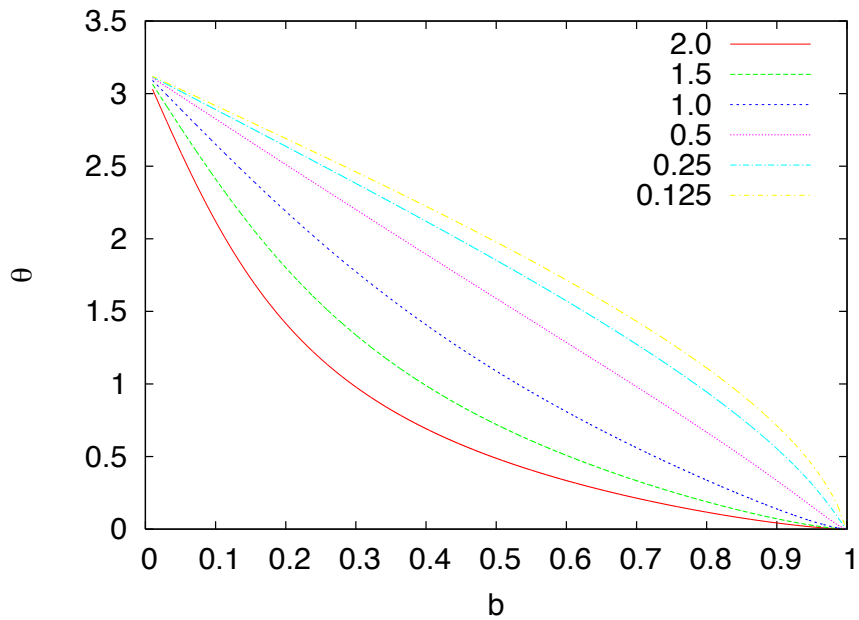



Figure 5.13: The impact parameter $b(\theta)$ for the potential given by equation (5.27) for different values of the initial velocity $v$. We set $a = 1$, $x(0) = -5$ and made 4000 integration steps from $t_i = 0$ to $t_f = 40$.

Another interesting dynamical field is given by the Yukawa potential. This is a phenomenological model of nuclear interactions:

$$V(r) = k\frac{\mathrm{e}^{-r/a}}{r}\,. \tag{5.28}$$

This field can also be used as a model of the effective interaction of electrons in metals (Thomas–Fermi) or as the Debye potential in a classic plasma. The resulting force is

$$\vec{F}(r) = f(r)\,\hat{r}\,, \qquad f(r) = k\frac{\mathrm{e}^{-r/a}}{r^2}\left(1+\frac{r}{a}\right) \tag{5.29}$$

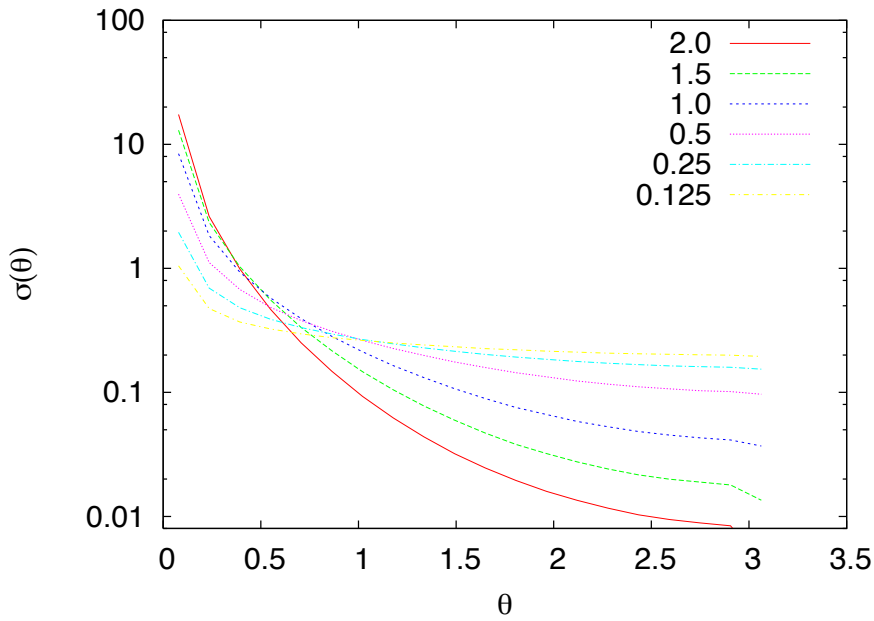



Figure 5.14: The function $\sigma(\theta)$ for the potential given by equation (5.27) for different values of the initial velocity $v$. We set $a = 1$, $x(0) = -5$ and the integration is performed by making 4000 steps from $t_i = 0$ to $t_f = 40$.

The program of the resulting acceleration can be found in the file `rk2_yu.f90`. The results are shown in figures 5.15–5.16.

## 5.5 More Particles

In this section we will generalize the discussion of the previous paragraphs in the case of a dynamical system with more degrees of freedom. The number of dynamical equations that need to be solved depends on the number of degrees of freedom and we have to write a program that implements the 4th order Runge–Kutta method for an arbitrary number of equations `NEQ`. We will explain how to allocate memory *dynamically*, in which case the necessary memory storage space, which depends on `NEQ`, is allocated at the time of running the program and not at compilation time.

Until now, memory has been allocated *statically*. This means that arrays have sizes which are known at compile time. For example, in the program `rk2.f90` the integer parameter `P` had a given value which determined the size of all arrays using the declarations:

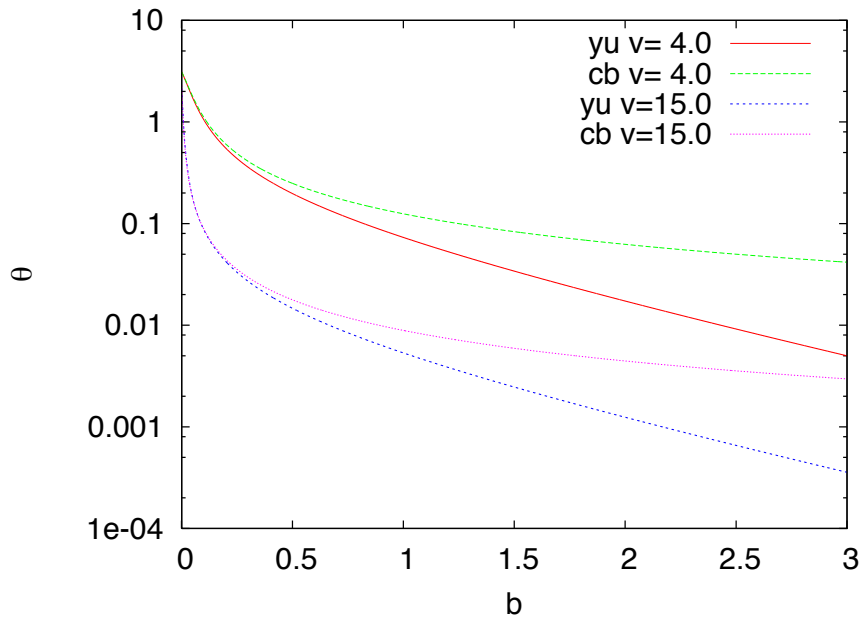



Figure 5.15: The function $b(\theta)$ for the Yukawa scattering for several values of the initial velocity $v$. We set $a = 1$, $k = 1$, $x(0) = -50$ and the integration is performed with 5000 steps from $t_i = 0$ to $t_f = 30$. The lines marked as cb are equation (5.24) of the Rutherford scattering.

```
integer,parameter     :: P=1010000
real(8),dimension(P):: T,X1,X2,V1,V2
```

Changing P after compilation is impossible and if this becomes necessary we have to edit the file, change the value of P and recompile. Dynamical memory allocation allows us to read in Nt and NEQ at execution time and *then* ask from the operating system to allocate the necessary memory. All we have to do is to declare the *shape* of the arrays (i.e. how many indices they take) and give them the allocatable attribute. The needed memory can be asked for at execution time by calling the function ALLOCATE. Here is an example:

```
integer Nt,NEQ
real(8),allocatable :: T (:)    ! Rank-1 array
real(8),allocatable :: X (:,:)  ! Rank-2 array
real(8),allocatable :: X0(:)    ! Rank-1 array

read *,Nt
```

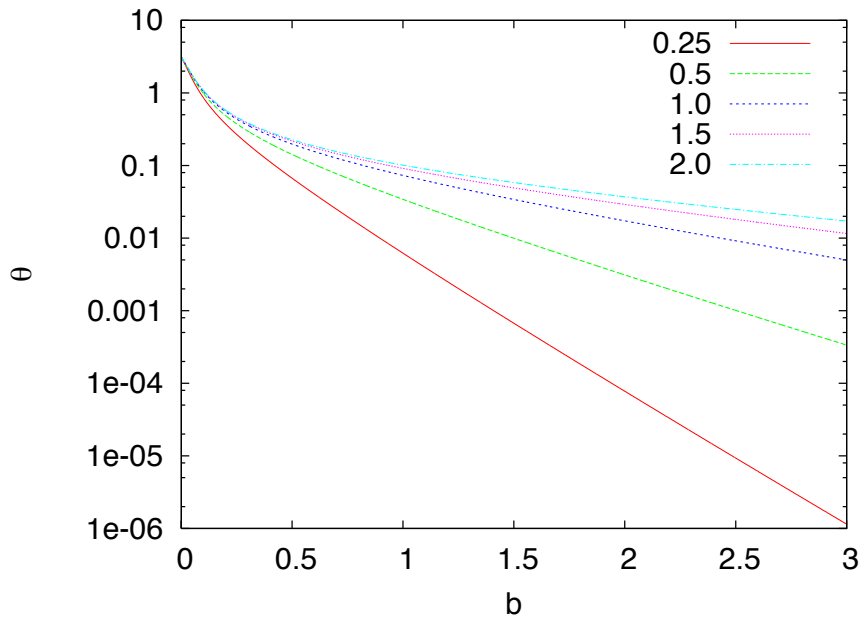



Figure 5.16: The function $b(\theta)$ for the Yukawa scattering for several values of the range $a$ of the force. We set $v = 4.0$, $k = 1$, $x(0) = -50$ and the integration is performed with 5000 steps from $t_i = 0$ to $t_f = 30$.

```
 call finit(NEQ)
 allocate(X0(NEQ))
 allocate(T(Nt))
 allocate(X(Nt,NEQ))
 ...
 (compute with X0,T,X)
 ...
 deallocate(X0)
 deallocate(X )
 deallocate(T )
 (X0,T,X are not usable anymore)
 ...
....................
subroutine finit(NEQ)
 NEQ = 4
end subroutine finit
```

In this program the arrays have the allocatable attribute and for each : they have an extra index. Therefore the arrays `T,X0` are rank-1 arrays and have only one index, whereas the array `X` is a rank-2 array and has two indices. The user enters the value of `Nt` and the subroutine `finit`

sets the value of NEQ. The calls to the function ALLOCATE allocate the necessary memory[4]. If memory allocation is successful, then the arrays can be used in the same way as the statically allocated ones. When allocatable arrays are not necessary anymore we should make a call to the function DEALLOCATE which returns the unused memory back to the system. Otherwise our program might suffer from "memory leaks" if e.g. the memory is repeatedly asked in a loop that calls a function that allocates memory without deallocating it in the end. Dynamical memory allocation is very convenient but for high performance computing static allocation might be preferable so that the compiler performs a more efficient optimization.

The main program will be written in the file rkA.f90, whereas the force-dependent part of the code will be written in files with names of the form rkA_XXX.f90. In the latter, the user must program a *subroutine* f(t,X,dXdt) which takes as input the time t and the values of the functions X(NEQ) and outputs the values of their derivatives dXdt(NEQ) at time t. The function finit(NEQ) sets the number of functions in f and it is called once during the initialization phase of the program.

The program in the file rkA.f90[5] is listed below:

```
!=========================================================
!Program to solve an ODE system using the
!4th order Runge-Kutta Method
!NEQ: Number of equations
!User supplies two subroutines:
!f(t,x,xdot): with real(8) :: t,x(NEQ),xdot(NEQ) which
!given the time t and current values of functions x(NEQ)
!it returns the values of derivatives: xdot = dx/dt
!The values of two coupling constants k1,k2 may be used
!in f which are read in the main program and stored in
!common /couplings/k1,k2
!finit(NEQ) : sets the value of NEQ
!
!User Interface:
!k1,k2: real(8) coupling constants
!Nt,Ti,Tf: Nt-1 integration steps, initial/final time
!X0: real(8),dimension(NEQ): initial conditions
!Output:
```

[4]We assume that Nt, NEQ are positive and small enough so that the requested memory is available. It is better to use the call allocate(T(Nt),STAT=IERR). The non zero value of IERR after the call indicates a successful allocation and the following test stops the program otherwise: IF(IERR .eq. 0) STOP 'Memory allocation for T failed'

[5]In the accompanying software you will find the files rkN.f90 and rkN_XXX.f90 which show you how to write the same program using static memory allocation.

```
!rkA.dat with Nt lines consisting of: T(Nt),X(Nt,NEQ)
!=========================================================
program rk2_solve
 implicit none
 real(8),allocatable :: T (:)
 real(8),allocatable :: X (:,:)
 real(8),allocatable :: X0(:)
 real(8) :: Ti,Tf
 integer :: Nt, NEQ,i
 real(8) ::        k1,k2
 common /couplings/k1,k2
!We need explicit interface, since energy has
!assumed-shape arrays as arguments.
 INTERFACE
  real(8) function energy(t_intrf,x_intrf)
   implicit none
   real(8) :: t_intrf,x_intrf(:)
  end function energy
 END INTERFACE
!Input:
 print *,'Runge-Kutta Method for ODE Integration.'
!Get the number of equations:
 call finit(NEQ);allocate(X0(NEQ))
 print *,'NEQ= ',NEQ
 print *,'Enter coupling constants:'
 read  *, k1,k2
 print *,'k1= ',k1,' k2= ',k2
 print *,'Enter Nt,Ti,Tf,X0:'
 read  *,       Nt,Ti,TF,X0
 print *,'Nt = ',Nt
 print *,'Time: Initial Ti =',Ti,' Final Tf=',Tf
 print '(A,2000G28.16)','              X0 =',X0
 allocate(T(Nt));allocate(X(Nt,NEQ))
!The Calculation:
 call RK(T,X,Ti,Tf,X0,Nt,NEQ)
!Output:
 open(unit=11,file='rkA.dat')
 do i=1,Nt
  write(11,'(2000G28.16)')T(i),X(i,:),&
      energy(T(i),X(i,:))
 enddo
 close(11)
end program rk2_solve
!=========================================================
!Driver of the RKSTEP routine
!=========================================================
subroutine RK(T,X,Ti,Tf,X0,Nt,NEQ)
 implicit none
 integer :: Nt,NEQ
```

```
 real(8),dimension(Nt)     :: T
 real(8),dimension(Nt,NEQ):: X
 real(8),dimension(NEQ)    :: X0
 real(8) :: Ti ,Tf
 real(8) :: dt
 real(8) :: TS,XS(NEQ) !values of time and X at given step
 integer :: i
!Initialize variables:
 dt      = (Tf-Ti)/(Nt-1)
 T (1)   = Ti
 X (1,:)= X0
 TS      = Ti
 XS      = X0
!Make RK steps: The arguments of RKSTEP are
!replaced with the new ones
 do i=2,Nt
  call RKSTEP(TS,XS,dt,NEQ)
  T(i)   = TS
  X(i,:)= XS
 enddo
end subroutine RK
!=========================================================
!Subroutine RKSTEP(t,X,dt)
!Runge-Kutta Integration routine of ODE
!=========================================================
subroutine RKSTEP(t,x,dt,NEQ)
 implicit none
 integer :: NEQ
 real(8),dimension(NEQ) :: x
 real(8) :: t,dt,tt
 real(8),dimension(NEQ) :: k1,k2,k3,k4,xx
 real(8) :: h,h2,h6
!We need explicit interface, since f has assumed-shape
!arrays as arguments.
 INTERFACE
  subroutine f(t_intrf,x_intrf,xdot_intrf)
   implicit none
   real(8) :: t_intrf
   real(8),dimension(:):: x_intrf,xdot_intrf
  end subroutine f
 END INTERFACE

 h =dt        !h =dt, integration step
 h2=0.5D0*h   !h2=h/2
 h6=h/6.0D0   !h6=h/6

 call f(t ,x ,k1); xx = x + h2*k1; tt =t+h2
 call f(tt,xx,k2); xx = x + h2*k2; tt =t+h2
 call f(tt,xx,k3); xx = x + h *k3; tt =t+h
```

```
 call f(tt,xx,k4)

 t =t+h
 x =x +h6*(k1+2.0D0*(k2+k3)+k4)
end subroutine RKSTEP
```

Note the use of *array sections*:

```
 write(11,'(2000G28.16)')T(i),X(i,:)
 X(1,:)= X0
 X(i,:)= XS
```

The expression `X (1,:)` refers to the first row of the array `X`. The arrays `X0` and `X (1,:)` are *conformable* and we can assign the entries in `X (1,:)` equal to the entries in `X0`, i.e. `X(1,1)=X0(1), X(1,2)=X0(2), ... , X(1,NEQ)=X0(NEQ)` in only one statement `X(1,:)= X0`. Similarly the statement `write(...) X(i,:)` prints the whole `i`-th row of the array `X` whereas the statement `X(i,:)= XS` assigns `X(i,1)=XS(1), X(i,2)=XS(2), ... , X(i,NEQ)=XS(NEQ)`. Note the vector operations:

```
xx = x + h2* k1
x  = x + h6*(k1+2.0D0*(k2+k3)+k4)
```

which are equivalent to the following do loops

```
 do i=1,NEQ
  xx(i) = x(i) + h2* k1(i)
 enddo
 do i=1,NEQ
  x(i)  = x(i) + h6*(k1(i)+2.0D0*(k2(i)+k3(i))+k4(i))
 enddo
```

A few words in order to explain what is an `INTERFACE` block. Up to now we declared only the type of the functions in the calling program. When the arguments of the function are arrays for which we only know their shape and not their size (assumed-shape arrays), the compiler needs more information. We need to declare the arguments, their types and, in case they are arrays, their shapes as well. Each program that calls these functions should include an `INTERFACE` block which provides this information. For the functions `f` and `energy`, the corresponding `INTERFACE` block is

```
 INTERFACE
```

```
!-----------------------------------------------
  subroutine f(t_intrf,x_intrf,xdot_intrf)
   implicit none
   real(8) :: t_intrf
   real(8),dimension(:):: x_intrf,xdot_intrf
  end subroutine f
!-----------------------------------------------
  real(8) function energy(t_intrf,x_intrf)
   implicit none
   real(8) :: t_intrf,x_intrf(:)
  end function energy
!-----------------------------------------------
 END INTERFACE
```

You may create files like e.g. `interfaces.inc` with groups of `INTERFACE` blocks and include them in all subprograms that use them with the statement `include "interfaces.inc"`.

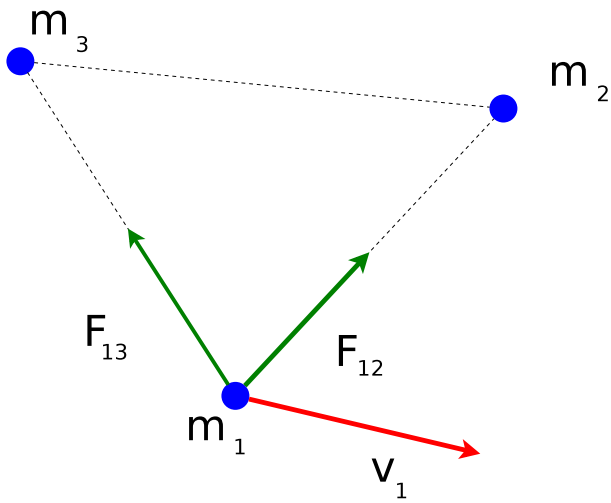



Figure 5.17: Three particles of equal mass interact via their mutual gravitational attraction. The problem is solved numerically using the program in the files `rkA.f90`, `rkA_3pcb.f90`. The same program can be used in order to study the motion of three equal charges under the influence of their attractive or repulsive electrostatic force.

Consider three particles of equal mass exerting a force of gravitational attraction on each other[6] like the ones shown in figure 5.17. The forces

[6]The same program can be used for three equal charges exerting an electrostatic force on each other, which can be either attractive or repulsive.

exerting on each other are given by

$$\vec{F}_{ij} = \frac{mk_1}{r_{ij}^3}\vec{r}_{ij}\,, \quad i,j = 1,2,3\,, \tag{5.30}$$

where $k_1 = -Gm$ and the equations of motion become $(i = 1,2,3)$

$$\begin{aligned} \frac{dx_i}{dt} &= v_{ix} \qquad \frac{dv_{ix}}{dt} = k_1 \sum_{j=1,j\neq i}^{3} \frac{x_i - x_j}{r_{ij}^3} \\ \frac{dy_i}{dt} &= v_{iy} \qquad \frac{dv_{iy}}{dt} = k_1 \sum_{j=1,j\neq i}^{3} \frac{y_i - y_j}{r_{ij}^3}\,, \end{aligned} \tag{5.31}$$

where $r_{ij}^2 = (x_i - x_j)^2 + (y_i - y_j)^2$. The total energy of the system is

$$E/m = \frac{1}{2}(v_1^2 + v_2^2) + \sum_{i,j=1,j<i}^{3} \frac{k_1}{r_{ij}}\,. \tag{5.32}$$

The relations shown above are programmed in the file `rkA_3pcb.f90` listed below:

```
!==============================
!Sets number of equations
!==============================
subroutine finit(NEQ)
 NEQ = 12
end subroutine finit
!==============================
!Three particles of the same
!mass on the plane interacting
!via Coulombic force
!==============================
subroutine f(t,X,dXdt)
 implicit none
 real(8) ::        k1,k2
 common /couplings/k1,k2
 real(8) :: t,X(:),dXdt(:)
!-------------------------
 real(8) :: x11,x12,x21,x22,x31,x32
 real(8) :: v11,v12,v21,v22,v31,v32
 real(8) :: r12,r13,r23
!-------------------------
 x11 = X(1);x21 = X(5);x31 = X(9)
 x12 = X(2);x22 = X(6);x32 = X(10)
```

```
 v11 = X(3);v21 = X(7);v31 = X(11)
 v12 = X(4);v22 = X(8);v32 = X(12)
!-----------------------
 r12 = ((x11-x21)*(x11-x21)+(x12-x22)*(x12-x22))**(-1.5D0)
 r13 = ((x11-x31)*(x11-x31)+(x12-x32)*(x12-x32))**(-1.5D0)
 r23 = ((x21-x31)*(x21-x31)+(x22-x32)*(x22-x32))**(-1.5D0)
!-------------
 dXdt(1)  = v11
 dXdt(2)  = v12
 dXdt(3)  = k1*(x11-x21)*r12+k1*(x11-x31)*r13 ! a11=dv11/dt
 dXdt(4)  = k1*(x12-x22)*r12+k1*(x12-x32)*r13 ! a12=dv12/dt
!-------------
 dXdt(5)  = v21
 dXdt(6)  = v22
 dXdt(7)  = k1*(x21-x11)*r12+k1*(x21-x31)*r23 ! a21=dv21/dt
 dXdt(8)  = k1*(x22-x12)*r12+k1*(x22-x32)*r23 ! a22=dv22/dt
!-------------
 dXdt(9)  = v31
 dXdt(10) = v32
 dXdt(11) = k1*(x31-x11)*r13+k1*(x31-x21)*r23 ! a31=dv31/dt
 dXdt(12) = k1*(x32-x12)*r13+k1*(x32-x22)*r23 ! a32=dv32/dt
end subroutine f
!==============================
real(8) function energy(t,X)
 implicit none
 real(8) ::        k1,k2
 common /couplings/k1,k2
 real(8) :: t,X(:)
!-----------------------
 real(8) :: x11,x12,x21,x22,x31,x32
 real(8) :: v11,v12,v21,v22,v31,v32
 real(8) :: r12,r13,r23
!-----------------------
 x11 = X(1);x21 = X(5);x31 = X(9)
 x12 = X(2);x22 = X(6);x32 = X(10)
 v11 = X(3);v21 = X(7);v31 = X(11)
 v12 = X(4);v22 = X(8);v32 = X(12)
!-----------------------
 r12 = ((x11-x21)*(x11-x21)+(x12-x22)*(x12-x22))**(-0.5D0)
 r13 = ((x11-x31)*(x11-x31)+(x12-x32)*(x12-x32))**(-0.5D0)
 r23 = ((x21-x31)*(x21-x31)+(x22-x32)*(x22-x32))**(-0.5D0)
!-----------------------
 energy = 0.5D0*&
         (v11*v11+v12*v12+v21*v21+v22*v22+v31*v31+v32*v32)
 energy = energy + k1*(r12+r13+r23)
end function energy
```

In order to run the program and see the results look at the commands in the shell script in the file `rkA_3pcb.csh`. In order to run the script use

command

```
> rkA_3pcb.csh −0.5 4000 1.5 −1 0.1 1 0 1 −0.1 −1 0 0.05 1 0 −1
```

which will run the program setting $k_1 = -0.5$, $\vec{r}_1(0) = -\hat{x}+0.1\hat{y}$, $\vec{v}_1(0) = \hat{x}$, $\vec{r}_2(0) = \hat{x} - 0.1\hat{y}$, $\vec{v}_2(0) = -\hat{x}$, $\vec{r}_3(0) = 0.05\hat{x} + \hat{y}$, $\vec{v}_3(0) = -\hat{y}$, `Nt`$= 4000$ and $t_f = 1.5$.

## 5.6 Problems

5.1 Reproduce the results shown in figures 5.3 and 5.4. Compare your results to the known analytic solution.

5.2 Write a program for the force on a charged particle in a constant magnetic field $\vec{B} = B\hat{k}$ and compute its trajectory for $\vec{v}(0) = v_{0x}\hat{x} + v_{0y}\hat{y}$. Set $x(0) = 1, y(0) = 0, v_{0y} = 0$ and calculate the resulting radius of the trajectory. Plot the relation between the radius and $v_{0x}$. Compare your results to the known analytic solution. (assume non relativistic motion)

5.3 Consider the anisotropic harmonic oscillator $a_x = -\omega_1^2 x$, $a_y = -\omega_2^2 y$. Construct the Lissajous curves by setting $x(0) = 0, y(0) = 1, v_x(0) = 1, v_y(0) = 0$, $t_f = 2\pi$, $\omega_2^2 = 1$, $\omega_1^2 = 1, 2, 4, 9, 16, \ldots$. What happens when $\omega_1^2 \neq n\omega_2^2$?

5.4 Reproduce the results displayed in table 5.1 and figures 5.5 and 5.6. Plot $\ln a$ vs $\ln T$ and calculate the slope of the resulting straight line by using the linear least squares method. Is it what you expect? Calculate the intercept and compare your result with the expected one.

5.5 Calculate the angular momentum with respect to the center of the force at each integration step of the planetary motion and check whether it is conserved. Show analytically that conservation of angular momentum implies that the position vector sweeps areas at constant rate.

5.6 Calculate the escape velocity of a planet $v_e$ for $GM = 10.0$, $y(0) = 0.0$, $x_0 = x(0) = 1$ using the following steps: First show that $v_0^2 = -GM(1/a) + v_e^2$. Then set $v_x(0) = 0$, $v_y(0) = v_0$. Vary $v_y(0) = v_0$ and measure the resulting semi-major axis $a$. Determine the intercept of the resulting straight line in order to calculate $v_e$.

5.7 Repeat the previous problem for $x_0 = 0.5$, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0. From the $v_e = f(1/x_0)$ plot confirm the relation (5.14).

5.8 Check that for the bound trajectory of a planet with $GM = 10.0$, $x(0) = 1$, $y(0) = 0.0$, $v_x(0) = 0$ , $v_y(0) = 4$ you obtain that $F_1P + F_2P = 2a$ for each point $P$ of the trajectory. The point $F_1$ is the center of the force. After determining the semi-major axis $a$ numerically, the point $F_2$ will be taken symmetric to $F_1$ with respect to the center of the ellipse.

5.9 Consider the planetary motion studied in the previous problem. Apply a momentary push in the tangential direction after the planet has completed 1/4 of its elliptical orbit. How stable is the particle trajectory (i.e. what is the dependence of the trajectory on the magnitude and the duration of the push?)? Repeat the problem when the push is in the vertical direction.

5.10 Consider the scattering potential of the positron-hydrogen system given by equation (5.26). Plot the functions $f(r)$ and $V(r)$ for different values of $a$. Calculate the total cross section $\sigma_{tot}$ numerically and show that it is equal to $\pi a^2$.

5.11 Consider the Morse potential of diatomic molecules:

$$V(r) = D\left(\exp(-2\alpha r) - 2\exp(-\alpha r)\right) \tag{5.33}$$

where $D, \alpha > 0$. Compute the solutions of the problem numerically in one dimension and compare them to the known analytic solutions when $E < 0$:

$$x(t) = \frac{1}{\alpha}\ln\left\{\frac{D - \sqrt{D(D-|E|)}\sin(\alpha t\sqrt{2|E|/m} + C)}{|E|}\,,\right\} \tag{5.34}$$

where the integration constant as a function of the initial position and energy is given by

$$C = \sin^{-1}\left[\frac{D - |E|e^{\alpha x_0}}{\sqrt{D(D-|E|)}}\right]. \tag{5.35}$$

We obtain a periodic motion with an energy dependent period $= (\pi/\alpha)\sqrt{2m/|E|}$. For $E > 0$ we obtain

$$x(t) = \frac{1}{\alpha}\ln\left\{\frac{\sqrt{D(D+E)}\cosh(\alpha t\sqrt{2E/m} + C) - D}{|E|}\right\} \tag{5.36}$$

whereas for $E = 0$

$$x(t) = \frac{1}{\alpha}\ln\left\{\frac{1}{2} + \frac{D\alpha^2}{m}(t+C)^2\right\}. \tag{5.37}$$

In these equations, the integration constant $C$ is given by a different relation and not by equation (5.35). Compute the motion in phase space $(x, \dot{x})$ and study the transition from open to closed trajectories.

5.12 Consider the effective potential term $V_{eff}(r) = l^2/2mr^2$ ($l \equiv |\vec{L}|$) in the previous problem. Plot the function $V_{tot}(r) = V(r) + V_{eff}(r)$ for $D = 20$, $\alpha = 1$, $m = 1$, $l = 1$, and of course for $r > 0$. Determine the equilibrium position and the ionization energy.

Calculate the solutions $x(t)$, $y(t)$, $y(x)$, $r(t)$ on the plane for $E > 0$, $E = 0$, and $E < 0$ numerically. In the $E < 0$ case consider the scattering problem and calculate the functions $b(\theta)$, $\sigma(\theta)$ and the total cross section $\sigma_{tot}$.

5.13 Consider the potential of the molecular model given by the force $\vec{F}(r) = f(r)\,\hat{r}$ where $f(r) = 24(2/r^{13} - 1/r^7)$. Calculate the potential $V(r)$ and plot the function $V_{tot}(r) = V(r) + V_{eff}(r)$. Determine the equilibrium position and the ionization energy.

Consider the problem of scattering and calculate $b(\theta)$, $\sigma(\theta)$ and $\sigma_{tot}$ numerically. How much do your results depend on the minimum scattering angle?

5.14 Compute the trajectories of a particle under the influence of a force $\vec{F} = -k/r^3\hat{r}$. Determine appropriate initial conditions that give a spiral trajectory.

5.15 Compute the total cross section $\sigma_{tot}$ for the Rutherford scattering both analytically and numerically. What happens to your numerical results as you vary the integration limits?

5.16 Write a program that computes the trajectory of a particle that moves on the plane in the static electric field of $N$ static point charges.

5.17 Solve the three body problem described in the text in the case of three different electric charges by making the appropriate changes to the program in the file `rkA_3cb.f90`.

5.18 Two charged particles of equal mass and charge are moving on the $xy$ plane in a constant magnetic field $\vec{B} = B\hat{z}$. Solve the equations of motion using a 4th order Runge–Kutta Method. Plot the resulting trajectories for the initial conditions that you will choose.

5.19 Three particles of equal mass $m$ are connected by identical springs. The springs' spring constant is equal to $k$ and their equilibrium length is equal to $l$. The particles move without friction on a horizontal plane. Solve the equations of motion of the system numerically by using a 4th order Runge–Kutta Method. Plot the resulting

trajectories for the initial conditions that you will choose. (Hint: Look in the files `rkA_3hoc.f90`, `rkA_3hoc.csh`.)

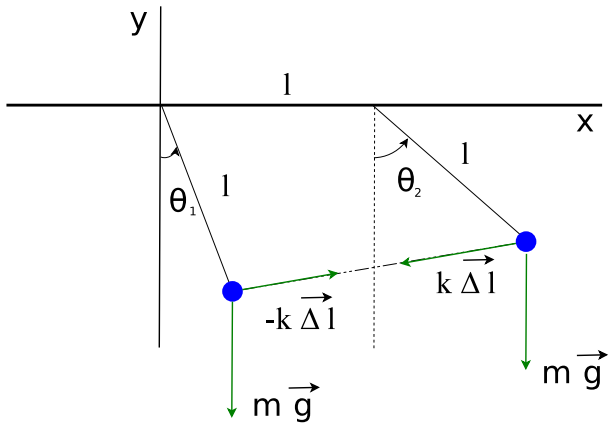



Figure 5.18: Two identical particles are attached to thin weightless rods of length $l$ and they are connected by an ideal weightless spring with spring constant $k$ and equilibrium length $l$. The rods are hinged to the ceiling at points whose distance is $l$. (Problem 5.20).

5.20 Two identical particles are attached to thin weightless rods of length $l$ and they are connected by an ideal weightless spring with spring constant $k$ and equilibrium length $l$. The rods are hinged to the ceiling at points whose distance is $l$ (see figure 5.18). Compute the Lagrangian of the system and the equations of motion for the degrees of freedom $\theta_1$ and $\theta_2$. Solve these equations numerically by using a 4th order Runge–Kutta method. Plot the positions of the particles in a Cartesian coordinate system and the resulting trajectory. Study the normal modes for small angles $\theta_1 \lesssim 0.1$ and compute the deviation of the solutions from the small oscillation approximation as the angles become larger. (Hint: Look in the files `rk_cpend.f90`, `rk_cpend.csh`)

5.21 Repeat the previous problem when the hinges of the rods slide without friction on the $x$ axis.

5.22 Repeat problem 5.20 by adding a third pendulum to the right at distance $l$.

# Chapter 6

# Motion in Space

In this chapter we will study the motion of a particle in space (three dimensions). We will also discuss the case of the relativistic motion, which is important if one wants to consider the motion of particles moving with speeds comparable to the speed of light. This will be an opportunity to use an *adaptive stepsize* Runge-Kutta method for the numerical solution of the equations of motion. We will use the open source code `rksuite`[1] available at the Netlib[2] repository. Netlib is an open source, high quality repository for numerical analysis software. The software it contains is used by many researchers in their high performance computing programs and it is a good investment of time to learn how to use it.

The technical skill that you will exercise in this chapter is looking for solutions to your numerical problems provided by software written by others. It is important to be able to locate the optimal solution to your problem, find the relevant functions, read the software's documentation carefully and filter out the necessary information in order to call and link the functions to your program.

[1] R.W. Brankin, I. Gladwell, and L.F. Shampine, RKSUITE: a suite of Runge-Kutta codes for the initial value problem for ODEs, Softreport 92-S1, Department of Mathematics, Southern Methodist University, Dallas, Texas, U.S.A, 1992.

[2] `www.netlib.org`

## 6.1 Adaptive Stepsize Control for Runge–Kutta Methods

The three dimensional equation of motion of a particle is an initial value problem given by the equations (4.6)

$$\begin{aligned}
\frac{dx}{dt} &= v_x & \frac{dv_x}{dt} &= a_x(t, x, v_x, y, v_y, z, v_z) \\
\frac{dy}{dt} &= v_y & \frac{dv_z}{dt} &= a_y(t, x, v_x, y, v_y, z, v_z) \\
\frac{dz}{dt} &= v_z & \frac{dv_z}{dt} &= a_z(t, x, v_x, y, v_y, z, v_z)\,.
\end{aligned} \tag{6.1}$$

For its numerical solution we will use an adaptive stepsize Runge–Kutta algorithm for increased performance and accuracy. Adaptive stepsize is used in cases where one needs to minimize computational effort for given accuracy goal. The method frequently changes the time step during the integration process, so that it is set to be large through smooth intervals and small when there are abrupt changes in the values of the functions. This is achieved by exercising *error control* either by monitoring a conserved quantity or by computing the same solution using two different methods. In our case, two Runge-Kutta methods are used, one of order $p$ and one of order $p+1$, and the difference of the results is used as an estimate of the truncation error. If the error needs to be reduced, the step size is reduced and if it is satisfactorily small the step size is increased. For the details we refer the reader to [28]. Our goal is not to analyze and understand the details of the algorithm, but to learn how to find and use appropriate and high quality code written by others. The link `http://www.netlib.org/ode/` reads

```
lib  rksuite
alg  Runge-Kutta
for  initial value problem for first order ordinary ↩
   differential
     equations. A suite of codes for solving IVPs in ODEs. A
     choice of RK methods, is available. Includes an error
     assessment facility and a sophisticated stiffness checker.
     Template programs and example results provided.
     Supersedes RKF45, DDERKF, D02PAF.
ref  RKSUITE, Softreport 92-S1, Dept of Math, SMU, Dallas, ↩
   Texas
by   R.W. Brankin (NAG), I. Gladwell and L.F. Shampine (SMU)
lang Fortran
```

```
prec double
```

There, we learn that the package provides code for Runge–Kutta methods, whose source is open and written in the Fortran language. We also learn that the code is written for double precision variables, which is suitable for our problem. Last, but not least, we are also happy to learn that it is written by highly reputable people! We download the files `rksuite.f`, `rksuite.doc`, `details.doc`, `templates`, `readme`.

In order to link the subroutines provided by the suite to our program we need to read the documentation carefully. In the general case, documentation is available on the web (html, pdf, ...), bundled files with names like `README`, `INSTALL`, in whole directories with names like `doc/`, online help in `man` or `info` pages and finally in good old fashioned printed manuals. Good quality software is also well documented inside the source code files, something that is true for the software at hand.

In order to link the suite's subroutines to our program we need the following basic information:

- INPUT DATA: This is the necessary information that the program needs in order to perform the calculation. In our case, the minimal such information is the initial conditions, the integration time interval and the number of integration steps. The user should also provide the functions on the right hand side of (6.1). It might also be necessary to provide information about the desired accuracy goal, the scale of the problem, the hardware etc.

- OUTPUT DATA: This is the information on how we obtain the results of the calculation for further analysis. Information whether the calculation was successful and error free could also be provided.

- WORKSPACE: This is information on how we provide the necessary memory space used in the intermediate calculations. Such space needs to be provided by the user in programming languages where dynamical memory allocation is not possible, like in `Fortran 77`, and the size of workspace depends on the parameters of the calling program.

It is easy to install the software. All the necessary code is in one file `rksuite.f`. The file `rksuite.doc`[3] contains the documentation. There we read that we need to inform the program about the hardware dependent

[3]This is a simple text file which you can read with the command `less rksuite.doc` or with `emacs`.

accuracy of floating point numbers. We need to set the values of three variables:

```
...
RKSUITE requires three environmental constants OUTCH, MCHEPS,
DWARF. When you use RKSUITE, you may need to know their
values. You can obtain them by calling the subroutine ENVIRN
in the suite:

  CALL ENVIRN(OUTCH,MCHPES,DWARF)

returns values

  OUTCH  - INTEGER
           Standard output channel on the machine being used.
  MCHEPS - DOUBLE PRECISION
           The unit of roundoff, that is, the largest
           positive number such that 1.0D0 + MCHEPS = 1.0D0.
  DWARF  - DOUBLE PRECISION
           The smallest positive number on the machine being
           used.
...
************************* Installation Details ************

 All machine-dependent aspects of the suite have been
 isolated in the subroutine ENVIRN in the rksuite.for file.
 Certain environmental parameters must be specified in this
 subroutine.  The values in the distribution version are
 those appropriate to the IEEE arithmetic standard.  They
 must be altered, if necessary, to values appropriate to the
 computing system you are using before calling the codes of
 the suite. If the IEEE arithmetic standard values are not
 appropriate for your system, appropriate values can often
 be obtained by calling routines named in the Comments of
 ENVIRN.
...
```

The variables `OUTCH, MCHEPS, DWARF` are defined in the subroutine `ENVIRN`. They are given generic default values but the programmer is free to change them by editing `ENVIRN`. We should identify the routine in the file `rksuite.f` and read the comments in it[4]:

```
...
      SUBROUTINE ENVIRN(OUTCH,MCHEPS,DWARF)
...
```

[4]These are lines that begin with a `C` as this is old fixed format Fortran code.

```
C  The following six statements are to be Commented out
C  after verification that the machine and installation
C  dependent quantities are specified correctly.
...
      WRITE(*,*) ' Before using RKSUITE, you must verify that the  '
      WRITE(*,*) ' machine- and installation-dependent quantities  '
      WRITE(*,*) ' specified in the subroutine ENVIRN are correct, '
      WRITE(*,*) ' and then Comment these WRITE statements and the '
      WRITE(*,*) ' STOP statement out of ENVIRN.                   '
      STOP
...
C  The following values are appropriate to IEEE
C  arithmetic with the typical standard output channel.
C
      OUTCH = 6
      MCHEPS = 1.11D-16
      DWARF = 2.23D-308
```

All we need to do is to comment out the `WRITE` and `STOP` commands since we will keep the default values of the `OUTCH`, `MCHEPS`, `DWARF` variables:

```
...
C     WRITE(*,*) ' Before using RKSUITE, you must verify that the  '
C     WRITE(*,*) ' machine- and installation-dependent quantities  '
C     WRITE(*,*) ' specified in the subroutine ENVIRN are correct, '
C     WRITE(*,*) ' and then Comment these WRITE statements and the '
C     WRITE(*,*) ' STOP statement out of ENVIRN.                   '
C     STOP
...
```

In order to check whether the default values are satisfactory, we can use the Fortran intrinsic functions `EPSILON()` and `TINY()`. In the file `test_envirn.f90`, we write a small test program

```
program testme
 implicit none
 integer ::  OUTCH
 real(8) ::  DWARF, MCHEPS
 real(8) ::  x

 OUTCH   = 6 !This is pretty much a standard
 MCHEPS  = epsilon(x)/2.0D0
 DWARF   = tiny(x)
 write(6,101)OUTCH,MCHEPS,DWARF

101 format(I4,2E30.18)
end program testme
```

We compile and run the above program as follows:

```
> gfortran test_envirn.f90  -o test_envirn
```

```
> ./test_envirn
   6       0.1110223024625156 54E-15       ↩
      0.222507385850720138-307
```

We conclude that our choices are satisfactory.

Next we need to learn how to use the subroutines in the suite. By carefully reading `rksuite.doc` we learn the following: The interface to the adaptive stepsize Runge–Kutta algorithm is the routine `UT` (`UT` = "Usual Task"). The routine can use a 2nd-3rd (RK23) order Runge-Kutta pair for error control (`METHOD=1`), a 4th-5th (RK45) order pair (`METHOD=2`) or a 7th-8th (RK78) order pair (`METHOD=3`). We will set `METHOD=2` (RK45). The routine `SETUP` must be called before `UT` for initialization. The user should provide a function `F` that calculates the derivatives of the functions we integrate for, i.e. the right hand side of 6.1.

The fastest way to learn how to use the above routines is "by example". The suite include a templates package which can be unpacked by executing the commands in the file `templates` using the `sh` shell:

```
> sh templates
tmpl1.out
tmpl1a.f
...
```

The file `tmpl1a.f` contains the solution of the simple harmonic oscillator and has many explanatory comments in it. We encourage the reader to study it carefully, run it and test its results.

After we become wise enough, we write the driver for the integration routine `UT`, which can be found in the file `rk3.f90`:

```
!=========================================================
!Program to solve a 6 ODE system using Runge-Kutta Method
!Output is written in file rk3.dat
!=========================================================
program rk3_solve
 include 'rk3.inc'
 real(8) :: T0,TF,X10,X20,X30,V10,V20,V30
 real(8) :: t,dt,tstep
 integer :: STEPS
 integer :: i
 real(8) :: energy
!Arrays/variables needed by rksuite:
 real(8) TOL,THRES(NEQ),WORK(LENWRK),Y(NEQ),YMAX(NEQ),&
      YP(NEQ),YSTART(NEQ),HSTART
 logical  ERRASS, MESSAGE
```

```
 integer  UFLAG
!.. External Subroutines ..
 EXTERNAL          F, SETUP, STAT, UT
!Input:
 print *,'Runge-Kutta Method for 6-ODEs Integration'
 print *,'Enter coupling constants k1,k2,k3,k4:'
 read  *, k1,k2,k3,k4
 print *,'k1= ',k1,' k2= ',k2,' k3= ',k3,' k4= ',k4
 print *,'Enter STEPS,T0,TF,X10,X20,X30,V10,V20,V30:'
 read  *, STEPS,T0,TF,X10,X20,X30,V10,V20,V30
 print *,'No. Steps= ',STEPS
 print *,'Time: Initial T0 =',T0,' Final TF=',TF
 print *,'            X1(T0)=',X10,' X2(T0)=',X20,' X3(T0)=',X30
 print *,'            V1(T0)=',V10,' V2(T0)=',V20,' V3(T0)=',V30
!Initial Conditions
 dt    = (TF-T0)/STEPS
 YSTART(1) = X10
 YSTART(2) = X20
 YSTART(3) = X30
 YSTART(4) = V10
 YSTART(5) = V20
 YSTART(6) = V30
!
!  Set error control parameters.
!
 TOL = 5.0D-6
 do i = 1, NEQ
  THRES(i) = 1.0D-10
 enddo
 MESSAGE = .TRUE.
 ERRASS  = .FALSE.
 HSTART  = 0.0D0
!Initialization:
 call SETUP(NEQ,T0,YSTART,TF,TOL,THRES,METHOD,'Usual Task',&
      ERRASS,HSTART,WORK,LENWRK,MESSAGE)
 open(unit=11,file='rk3.dat')
 write(11,100) T0,YSTART(1),YSTART(2),YSTART(3),YSTART(4),&
      YSTART(5),YSTART(6),energy(T0,YSTART)
!Calculation:
 do i=1,STEPS
  t = T0 + i*dt
  call UT(F,t,tstep,Y,YP,YMAX,WORK,UFLAG)
  if(UFLAG.GT.2) exit !exit the loop: go after enddo
  write(11,100) tstep,Y(1),Y(2),Y(3),Y(4),Y(5),Y(6),&
       energy(tstep,Y)
 enddo
 close(11)
100 format(8E25.15)
end program rk3_solve
```

All common parameters and variables are declared in an include file `rk3.inc`. This is necessary in order for them to be accessible by the function `F` which calculates the derivatives. The contents of this file are substituted in each line containing the command `include 'rk3.inc'`.

```
!Basic definitions of variables for the suite rksuite
 implicit none
!NEQ is the number of equations, 6 in 3 dimensions
!METHOD=2 is for RK45.
 INTEGER            NEQ,  LENWRK,        METHOD
 PARAMETER         (NEQ=6,LENWRK=32*NEQ,METHOD=2)
 REAL *8           k1,k2,k3,k4 !force couplings
 COMMON /COUPLINGS/k1,k2,k3,k4
```

The number of differential equations is set equal to `NEQ=6`. The integration method is set by the choice `METHOD=2`. The variable `LENWRK` sets the size of the workspace needed by the suite for the intermediate calculations.

The main program starts with the user interface. The initial state of the particle is stored in the array `YSTART` in the positions $1 \ldots 6$. The first three positions are the coordinates of the initial position and the last three the components of the initial velocity. Then we set some variables that determine the behavior of the integration program (see the file `rksuite.doc` for details) and call the subroutine `SETUP`. The main integration loop is:

```
 do i=1,STEPS
  t = T0 + i*dt
  call UT(F,t,tstep,Y,YP,YMAX,WORK,UFLAG)
  if(UFLAG.GT.2) exit !exit the loop: go after enddo
  write(11,100) tstep,Y(1),Y(2),Y(3),Y(4),Y(5),Y(6),&
       energy(tstep,Y)
 enddo
```

The function `F` is the subroutine that calculates the derivatives and it will be programmed by us later. The variable `t` stores the *desired* moment of time at which we want to calculate the functions. Because of the adaptive stepsize, it can be different than the one returned by the subroutine `UT`. The actual value of time that the next step lands[5] on is `tstep`. The array `Y` stores the values of the functions. We choose the data structure to be such that $x$= `Y(1)`, $y$= `Y(2)`, $z$= `Y(3)` and $v_x$= `Y(4)`, $v_y$= `Y(5)`, $v_z$= `Y(6)`

[5]When `UGLAG` $\leq 2$, `tstep=t` and we will not worry about them being different with our program.

(the same sequence as in the array `YSTART`). The function `energy(t,Y)` returns the value of the mechanical energy of the particle and its code will be written in the same file as that of `F`. Finally, the variable `UFLAG` indicates the error status of the calculation by `UT` and if `UFLAG`$> 2$ we end the calculation.

Our test code will be on the study of the motion of a projectile in a constant gravitational field, subject also to the influence of a dissipative force $\vec{F}_r = -mk\vec{v}$. The program is in the file `rk3_g.f90`. We choose the parameters `k1` and `k2` so that $\vec{g} =$ `-k1` $\hat{k}$ and $k =$ `k2`.

```
!-----------------------------------
subroutine F(T,Y,YP)
 include 'rk3.inc'
 real(8) :: t
 real(8) :: Y(*),YP(*)
 real(8) :: x1,x2,x3,v1,v2,v3
 x1 = Y(1);v1 = Y(4)
 x2 = Y(2);v2 = Y(5)
 x3 = Y(3);v3 = Y(6)
!Velocities:   dx_i/dt = v_i
 YP(1) = v1
 YP(2) = v2
 YP(3) = v3
!Acceleration: dv_i/dt = a_i
 YP(4) = -k2*v1
 YP(5) = -k2*v2
 YP(6) = -k2*v3-k1
end subroutine F
!-----------------------------------
real(8) function energy(T,Y)
 include 'rk3.inc'
 real(8) :: t,e
 real(8) :: Y(*)
 real(8) :: x1,x2,x3,v1,v2,v3
 x1 = Y(1);v1 = Y(4)
 x2 = Y(2);v2 = Y(5)
 x3 = Y(3);v3 = Y(6)
!Kinetic Energy
 e = 0.5*(v1*v1+v2*v2+v3*v3)
!Potential Energy
 e = e + k1*x3
 energy = e
end function energy
```

For convenience we "translated" the values in the array `Y(NEQ)` into

user-friendly variable names[6]. If the file `rksuite.f` is in the directory `rksuite/`, then the compilation, running and visualization of the results can be done with the commands:

```
> gfortran rk3.f90 rk3_g.f90 rksuite/rksuite.f -o rk3
> ./rk3
 Runge-Kutta Method for 6-ODEs Integration
 Enter coupling constants k1,k2,k3,k4:
10 0 0 0
 k1=    10.0000        k2=   0.0000E+000  k3=
  0.0000E+000  k4=   0.0000E+000
 Enter STEPS,T0,TF,X10,X20,X30,V10,V20,V30:
10000 0 3 0 0 0 1 1 1
 No. Steps=        10000
 Time: Initial T0 =  0.0000E+000  Final TF=  3.0000
            X1(T0)=  0.0000E+000    X2(T0)=  0.0000E+000
            X3(T0)=  0.0000E+000
            V1(T0)=  1.0000         V2(T0)=   1.0000
            V3(T0)=  1.0000
> gnuplot
gnuplot> plot "rk3.dat"  using 1:2 with lines title "x1(t)"
gnuplot> plot "rk3.dat"  using 1:3 with lines title "x2(t)"
gnuplot> plot "rk3.dat"  using 1:4 with lines title "x3(t)"
gnuplot> plot "rk3.dat"  using 1:5 with lines title "v1(t)"
gnuplot> plot "rk3.dat"  using 1:6 with lines title "v2(t)"
gnuplot> plot "rk3.dat"  using 1:7 with lines title "v3(t)"
gnuplot> plot "rk3.dat"  using 1:8 with lines title "E(t)"
gnuplot> set title "trajectory"
gnuplot> splot "rk3.dat" using 2:3:4 with lines  notitle
```

All the above commands can be executed together using the shell script in the file `rk3.csh`. The script uses the animation script `rk3_animate.csh`. The following command executes all the commands shown above:

```
./rk3.csh -f 1 -- 10 0. 0 0 0 0 0 1 1 1 10000 0 3
```

[6]Note the declaration of the arrays `Y, YP: real(8) :: Y(*),YP(*)`. These arrays are "assumed-size" arrays for the functions `F, energy`, i.e. arrays whose size is unknown to the procedure. For arrays of more than one dimension, only the last index is allowed to be `*`. In general it is recommended that assumed-size arrays be avoided and declare them as assumed-shape like in the program `rkA.f90` of page 273. The declaration in this case is `real(8) :: Y(:),YP(:)`

## 6.2 Motion of a Particle in an EM Field

In this section we study the non-relativistic motion of a charged particle in an electromagnetic (EM) field. The particle is under the influence of the Lorentz force:

$$\vec{F} = q(\vec{E} + \vec{v} \times \vec{B}) \,. \tag{6.2}$$

Consider the constant EM field of the form $\vec{E} = E_x\hat{x} + E_y\hat{y} + E_z\hat{z}$, $\vec{B} = B\hat{z}$. The components of the acceleration of the particle are:

$$\begin{aligned} a_x &= (qE_x/m) + (qB/m)v_y \\ a_y &= (qE_y/m) - (qB/m)v_x \\ a_z &= (qE_z/m) \,. \end{aligned} \tag{6.3}$$

This field is programmed in the file `rk3_B.f90`. We set `k1` $= qB/m$, `k2` $= qE_x/m$, `k3` $= qE_y/m$ and `k4` $= qE_z/m$:

```
!-----------------------------------------------------
!Particle in constant Magnetic and electric field
!q B/m = k1 z    q E/m = k2 x + k3 y + k4 z
subroutine F(T,Y,YP)
 include 'rk3.inc'
 real(8) :: t
 real(8) :: Y(*),YP(*)
 real(8) :: x1,x2,x3,v1,v2,v3
 x1 = Y(1);v1 = Y(4)
 x2 = Y(2);v2 = Y(5)
 x3 = Y(3);v3 = Y(6)
!Velocities:   dx_i/dt = v_i
 YP(1) = v1
 YP(2) = v2
 YP(3) = v3
!Acceleration: dv_i/dt = a_i
 YP(4) = k2 + k1 * v2
 YP(5) = k3 - k1 * v1
 YP(6) = k4
end subroutine F
!-----------------------------------
real(8) function energy(T,Y)
 include 'rk3.inc'
 real(8) :: t,e
 real(8) :: Y(*)
 real(8) :: x1,x2,x3,v1,v2,v3
 x1 = Y(1);v1 = Y(4)
 x2 = Y(2);v2 = Y(5)
 x3 = Y(3);v3 = Y(6)
```

```
!Kinetic Energy
 e = 0.5*(v1*v1+v2*v2+v3*v3)
!Potential Energy
 e = e - k2*x1^2 - k3*x2 - k4*x3
 energy = e
end function energy
```

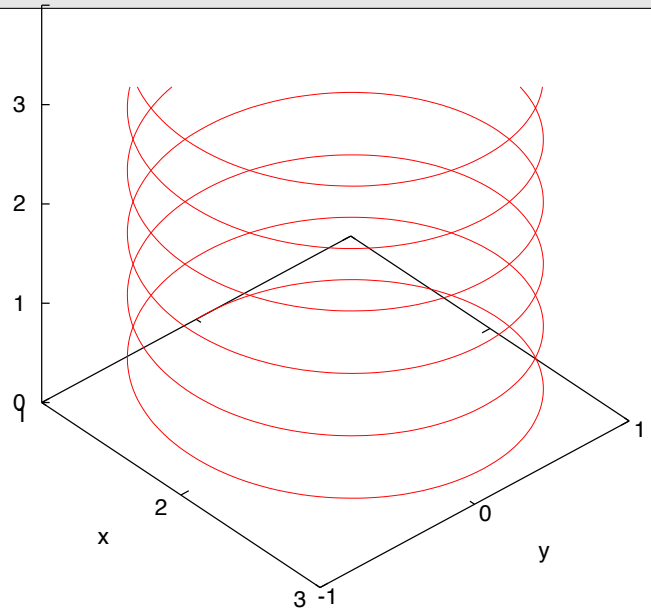



Figure 6.1: The trajectory of a charged particle in a constant magnetic field $\vec{B} = B\hat{z}$, where $qB/m = 1.0$, $\vec{v}(0) = 1.0\hat{y} + 0.1\hat{z}$, $\vec{r}(0) = 1.0\hat{x}$. The integration of the equations of motion is performed using the RK45 method from $t_0 = 0$ to $t_f = 40$ with 1000 steps.

We can also study space-dependent fields in the same way. The fields must satisfy Maxwell's equations. We can study the confinement of a particle in a region of space by a magnetic field by taking $\vec{B} = B_y\hat{y} + B_z\hat{z}$ with $qB_y/m = -k_2 y$, $qB_z/m = k_1 + k_2 z$ and $qB_y/m = k_3 z$, $qB_z/m = k_1 + k_2 y$. Note that $\vec{\nabla} \cdot \vec{B} = 0$. You may also want to calculate the current density from the equation $\vec{\nabla} \times \vec{B} = \mu_0 \vec{j}$.

The results are shown in figures 6.1–6.4.

## 6.3 Relativistic Motion

Consider a particle of non zero rest mass moving with speed comparable to the speed of light. In this case, it is necessary to study its motion using

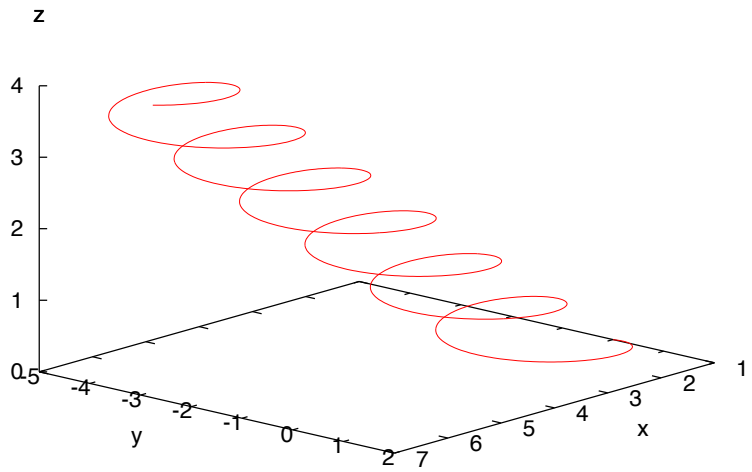



Figure 6.2: The trajectory of a charged particle in a constant magnetic field $\vec{B} = B\hat{z}$, where $qB/m = 1.0$ and a constant electric field $\vec{E} = E_x\hat{x}+E_y\hat{y}$ με $qE_x/m = qE_y/m = 0.1$. $\vec{v}(0) = 1.0\hat{y} + 0.1\hat{z}$, $\vec{r}(0) = 1.0\hat{x}$. The integration of the equations of motion is performed using the RK45 method from $t_0 = 0$ to $t_f = 40$ with 1000 steps. Each axis is on a different scale.

the equations of motion given by special relativity[7]. In the equations below we set $c = 1$. The particle's rest mass is $m_0 > 0$, its mass is $m = m_0/\sqrt{1-v^2}$ (where $v < 1$), its momentum is $\vec{p} = m\vec{v}$ and its energy is $E = m = \sqrt{p^2 + m_0^2}$. Then the equations of motion in a dynamic field $\vec{F}$ are given by:

$$\frac{d\vec{p}}{dt} = \vec{F}\,. \tag{6.4}$$

In order to write a system of first order equations, we use the relations

$$\vec{v} = \frac{\vec{p}}{m} = \frac{\vec{p}}{E} = \frac{\vec{p}}{\sqrt{p^2 + m_0^2}}\,. \tag{6.5}$$

[7]Of course for lower speeds, the special relativity equations of motion are a better approximation to the particle's motion, but the corrections to the non relativistic equations of motion are negligible.

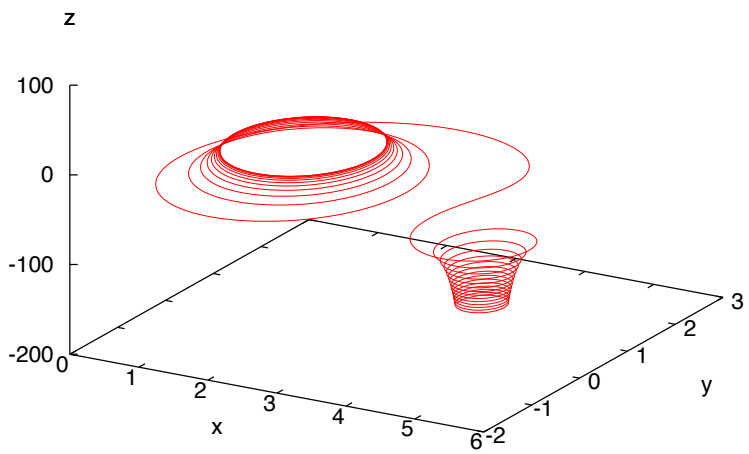



Figure 6.3: The trajectory of a charged particle in a magnetic field $\vec{B} = B_y\hat{y} + B_z\hat{z}$ with $qB_y/m = -0.02y$, $qB_z/m = 1 + 0.02z$, $\vec{v}(0) = 1.0\hat{y} + 0.1\hat{z}$, $\vec{r}(0) = 1.0\hat{x}$. The integration of the equations of motion is performed using the RK45 method from $t_0 = 0$ to $t_f = 500$ with 10000 steps. Each axis is on a different scale.

Using $\vec{v} = d\vec{r}/dt$ we obtain

$$\begin{aligned}
\frac{dx}{dt} &= \frac{(p_x/m_0)}{\sqrt{1+(p/m_0)^2}}, & \frac{d(p_x/m_0)}{dt} &= \frac{F_x}{m_0} \\
\frac{dy}{dt} &= \frac{(p_y/m_0)}{\sqrt{1+(p/m_0)^2}}, & \frac{d(p_y/m_0)}{dt} &= \frac{F_y}{m_0} \\
\frac{dz}{dt} &= \frac{(p_z/m_0)}{\sqrt{1+(p/m_0)^2}}, & \frac{d(p_z/m_0)}{dt} &= \frac{F_z}{m_0},
\end{aligned} \tag{6.6}$$

which is a system of first order differential equations for the functions $(x(t), y(t), z(t), (p_x/m_0)(t), (p_y/m_0)(t), (p_z/m_0)(t))$. Given the initial conditions $(x(0), y(0), z(0), (p_x/m_0)(0), (p_y/m_0)(0), (p_z/m_0)(0))$ their solution is unique and it can be computed numerically using the 4th-5th order Runge–Kutta method according to the discussion of the previous section.

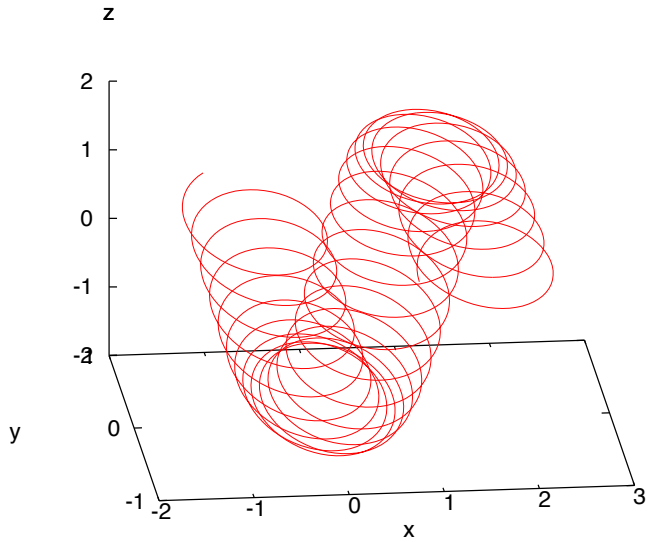



Figure 6.4: The trajectory of a charged particle in a magnetic field $\vec{B} = B_y\hat{y} + B_z\hat{z}$ with $qB_y/m = 0.08z$, $qB_z/m = 1.4 + 0.08y$, $\vec{v}(0) = 1.0\hat{y} + 0.1\hat{z}$, $\vec{r}(0) = 1.0\hat{x}$. The integration of the equations of motion is performed using the RK45 method from $t_0 = 0$ to $t_f = 3000$ with 40000 steps. Each axis is on a different scale.

By using the relations

$$
\begin{aligned}
(p_x/m_0) &= \frac{v_x}{\sqrt{1-v^2}} \qquad & v_x &= \frac{(p_x/m_0)}{\sqrt{1+(p/m_0)^2}} \\
(p_y/m_0) &= \frac{v_y}{\sqrt{1-v^2}} \qquad & v_y &= \frac{(p_y/m_0)}{\sqrt{1+(p/m_0)^2}} \\
(p_z/m_0) &= \frac{v_z}{\sqrt{1-v^2}} \qquad & v_z &= \frac{(p_z/m_0)}{\sqrt{1+(p/m_0)^2}}\,,
\end{aligned}
\tag{6.7}
$$

we can use the initial conditions $(x(0),\, y(0),\, z(0),\, v_x(0),\, v_y(0),\, v_z(0))$ instead. Similarly, from the solutions $(x(t),\, y(t),\, z(t),\, (p_x/m_0)(t),\, (p_y/m_0)(t),\, (p_z/m_0)(t))$ we can calculate $(x(t),\, y(t),\, z(t),\, v_x(t),\, v_y(t),\, v_z(t))$. We always have to check that

$$
v^2 = (v_x)^2 + (v_y)^2 + (v_z)^2 < 1\,. \tag{6.8}
$$

Since half of the functions that we integrate for are the momentum instead of the velocity components, we need to make some modifications to the

program in the file `rk3.f90`. The main program can be found in the file `sr.f90`:

```
!=========================================================
!Program to solve a 6 ODE system using Runge-Kutta Method
!Output is written in file sr.dat
!Interface to be used with relativistic particles.
!=========================================================
program sr_solve
 include 'sr.inc'
 real(8) :: T0,TF,X10,X20,X30,V10,V20,V30
 real(8) :: P10,P20,P30
 real(8) :: P1,P2,P3,V1,V2,V3
 real(8) :: t,dt,tstep
 integer :: STEPS
 integer :: i
 real(8) :: energy
!Arrays/variables needed by rksuite:
 real(8) :: TOL,THRES(NEQ), WORK(LENWRK), Y(NEQ), YMAX(NEQ),&
      YP(NEQ), YSTART(NEQ),HSTART
 logical :: ERRASS, MESSAGE
 integer :: UFLAG
!.. External Subroutines ..
 EXTERNAL          F, SETUP, STAT, UT
!Input:
 print *,'Runge-Kutta Method for 6-ODEs Integration'
 print *,'Special Relativistic Particle:'
 print *,'Enter coupling constants k1,k2,k3,k4:'
 read  *, k1,k2,k3,k4
 print *,'k1= ',k1,' k2= ',k2,' k3= ',k3,' k4= ',k4
 print *,'Enter STEPS,T0,TF,X10,X20,X30,V10,V20,V30:'
 read  *, STEPS,T0,TF,X10,X20,X30,V10,V20,V30
 call momentum(V10,V20,V30,P10,P20,P30)
 print *,'No. Steps= ',STEPS
 print *,'Time: Initial T0 =',T0,' Final TF=',TF
 print *,'           X1(T0)=',X10,' X2(T0)=',X20,' X3(T0)=',X30
 print *,'           V1(T0)=',V10,' V2(T0)=',V20,' V3(T0)=',V30
 print *,'           P1(T0)=',P10,' P2(T0)=',P20,' P3(T0)=',P30

!Initial Conditions
 dt    = (TF-T0)/STEPS
 YSTART(1) = X10
 YSTART(2) = X20
 YSTART(3) = X30
 YSTART(4) = P10
 YSTART(5) = P20
 YSTART(6) = P30
!
```

```
!  Set error control parameters.
!
 TOL = 5.0D-6
 do i = 1, NEQ
  THRES(i) = 1.0D-10
 enddo
 MESSAGE = .TRUE.
 ERRASS  = .FALSE.
 HSTART  = 0.0D0
!Initialization:
 call SETUP(NEQ,T0,YSTART,TF,TOL,THRES,METHOD,'Usual Task',&
      ERRASS,HSTART,WORK,LENWRK,MESSAGE)
 open(unit=11,file='sr.dat')
 call velocity(YSTART(4),YSTART(5),YSTART(6),V1,V2,V3)
 write(11,100) T0,YSTART(1),YSTART(2),YSTART(3),&
      V1,V2,V3,&
      energy(T0,YSTART),&
      YSTART(4),YSTART(5),YSTART(6)
!Calculation:
 do i=1,STEPS
  t = T0 + i*dt
  call UT(F,t,tstep,Y,YP,YMAX,WORK,UFLAG)
  if(UFLAG.GT.2) exit
  call velocity(Y(4),Y(5),Y(6),V1,V2,V3)
  write(11,100) tstep,Y(1),Y(2),Y(3),&
       V1,V2,V3,&
       energy(tstep,Y),&
       Y(4),Y(5),Y(6)
 enddo
 close(11)
100 format(11E25.15)
end program sr_solve
!=========================================================
!momentum -> velocity  transformation
!=========================================================
subroutine velocity(p1,p2,p3,v1,v2,v3)
 implicit none
 real(8) :: v1,v2,v3,p1,p2,p3,v,p,vsq,psq

 psq = p1*p1+p2*p2+p3*p3

 v1  = p1/sqrt(1.0D0+psq)
 v2  = p2/sqrt(1.0D0+psq)
 v3  = p3/sqrt(1.0D0+psq)
end subroutine velocity
!=========================================================
!velocity -> momentum transformation
!=========================================================
subroutine momentum(v1,v2,v3,p1,p2,p3)
```

```
 implicit none
 real(8) :: v1,v2,v3,p1,p2,p3,v,p,vsq,psq

 vsq = v1*v1+v2*v2+v3*v3
 if(vsq .ge. 1.0D0 ) stop 'sub momentum: vsq >= 1'
 p1  = v1/sqrt(1.0D0-vsq)
 p2  = v2/sqrt(1.0D0-vsq)
 p3  = v3/sqrt(1.0D0-vsq)
end subroutine momentum
```

The subroutines `momentum` and `velocity` compute the transformations (6.7). In the subroutine `momentum` we check whether the condition (6.8) is satisfied. These functions are also used in the subroutine `F` that computes the derivatives of the functions.

The test drive of the above program is the well known relativistic motion of a charged particle in a constant EM field. The acceleration of the particle is given by equations (6.3). The relativistic kinetic energy of the particle is

$$T = \left(\frac{1}{\sqrt{1-v^2}} - 1\right) m_0 = \left(\sqrt{1+(p/m_0)^2} - 1\right) m_0 \tag{6.9}$$

These relations are programmed in the file `sr_B.f90`. The contents of the file `sr_B.f90` are:

```
!=========================================================
!  Particle in constant Magnetic and electric field
!  q B/m = k1 z    q E/m = k2 x + k3 y + k4 z
!=========================================================
subroutine F(T,Y,YP)
 include 'sr.inc'
 real(8) :: t
 real(8) :: Y(*),YP(*)
 real(8) :: x1,x2,x3,v1,v2,v3,p1,p2,p3
 x1 = Y(1);p1 = Y(4)
 x2 = Y(2);p2 = Y(5)
 x3 = Y(3);p3 = Y(6)
 call velocity(p1,p2,p3,v1,v2,v3)
!now we can use all x1,x2,x3,p1,p2,p3,v1,v2,v3
 YP(1) = v1
 YP(2) = v2
 YP(3) = v3
!Acceleration:
 YP(4) = k2 + k1 * v2
 YP(5) = k3 - k1 * v1
 YP(6) = k4
```

```
end subroutine F
!=========================================================
!Energy per unit rest mass
!=========================================================
real(8) function energy(T,Y)
 include 'rk3.inc'
 real(8) :: t,e
 real(8) :: Y(*)
 real(8) :: x1,x2,x3,v1,v2,v3,p1,p2,p3,psq
 x1 = Y(1);p1 = Y(4)
 x2 = Y(2);p2 = Y(5)
 x3 = Y(3);p3 = Y(6)
 psq= p1*p1+p2*p2+p3*p3
!Kinetic Energy/m_0
 e = sqrt(1.0D0+psq)-1.0D0
!Potential Energy/m_0
 e = e - k2*x1 - k3*x2 - k4*x3
 energy = e
end function energy
```

The results are shown in figures 6.5–6.6.

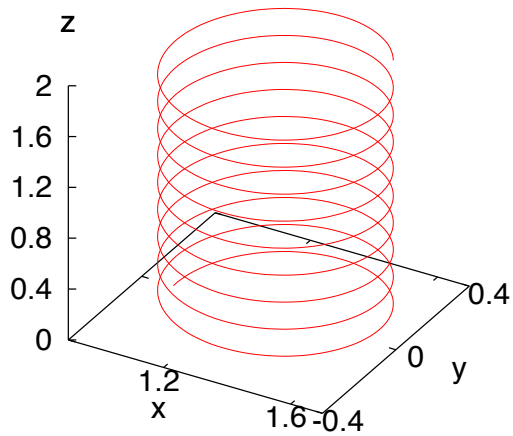



Figure 6.5: The trajectory of a relativistic charged particle in a magnetic field $\vec{B} = B_z\hat{z}$ with $qB_z/m_0 = 10.0$, $\vec{v}(0) = 0.95\hat{y} + 0.10\hat{z}$, $\vec{r}(0) = 1.0\hat{x}$. The integration is performed by using the RK45 method from $t_0 = 0$ to $t_f = 20$ with 1000 steps. Each axis is on a different scale.

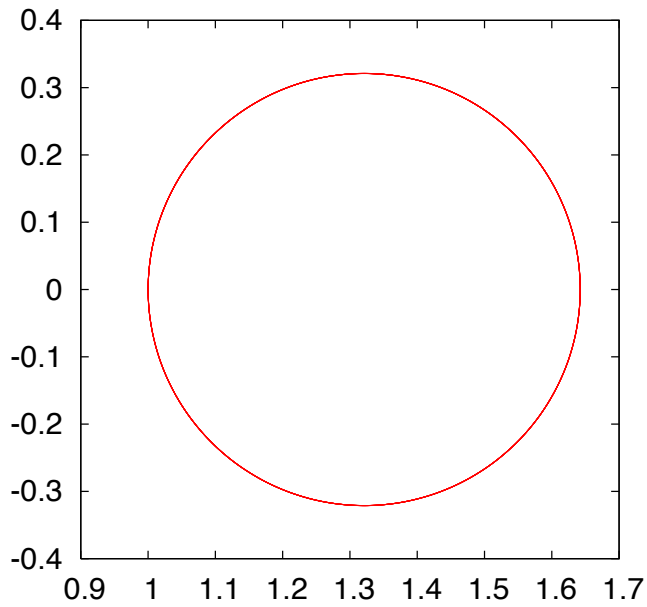



Figure 6.6: Projection of the trajectory of a relativistic charged particle in a magnetic field $\vec{B} = B_z\hat{z}$ with $qB_z/m_0 = 10.0$, on the $xy$ plane. $\vec{v}(0) = 0.95\hat{y} + 0.10\hat{z}$, $\vec{r}(0) = 1.0\hat{x}$. The integration is performed by using the RK45 method from $t_0 = 0$ to $t_f = 20$ with 1000 steps. Each axis is on a different scale.

Now we can study a more interesting problem. Consider a simple model of the Van Allen radiation belt. Assume that the electrons are moving within the Earth's magnetic field which is modeled after a magnetic dipole field of the form:

$$\vec{B} = B_0 \left(\frac{R_E}{r}\right)^3 \left[3(\hat{d}\cdot\hat{r})\,\hat{r} - \hat{d}\right] , \tag{6.10}$$

where $\vec{d} = d\hat{d}$ is the magnetic dipole moment of the Earth's magnetic field and $\vec{r} = r\hat{r}$. The parameter values are approximately equal to $B_0 = 3.5 \times 10^{-5}T$, $r \sim 2R_E$, where $R_E$ is the radius of the Earth. The typical energy of the moving particles is $\sim 1$ MeV which corresponds to velocities of magnitude $v/c = \sqrt{E^2 - m_0^2}/E \approx \sqrt{1 - 0.512^2}/1 = 0.86$. We choose the coordinate axes so that $\hat{d} = \hat{z}$ and we measure distance in $R_E$ units[8]. Then

[8]Since $c = 1$, the unit of time is the time that the light needs to travel distance equal to $R_E$ in the vacuum.

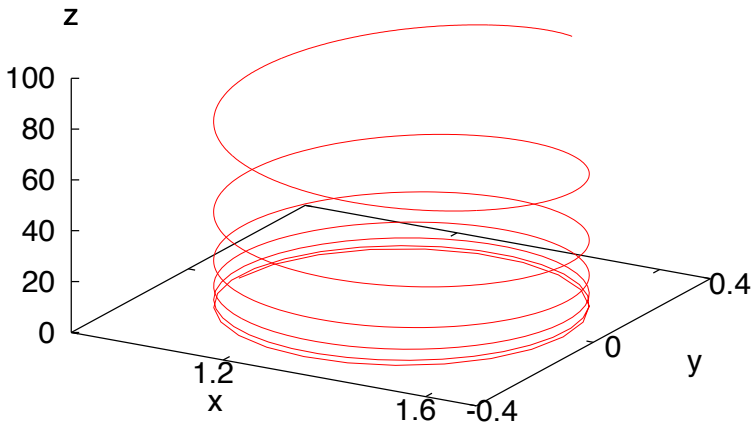



Figure 6.7: The influence of an additional electric field $q\vec{E}/m_0 = 1.0\hat{z}$ on the trajectory shown in figure 6.5.

we obtain:

$$
\begin{aligned}
B_x &= B_0 \frac{3xz}{r^5} \\
B_y &= B_0 \frac{3yz}{r^5} \\
B_z &= B_0 \left( \frac{3yz}{r^5} - \frac{1}{r^3} \right)
\end{aligned}
\tag{6.11}
$$

The magnetic dipole field is programmed in the file `sr_Bd.f90`:

```
!=========================================================
!  Particle in Magnetic dipole field:
!  q B_1/m = k1 (3 x1 x3)/r^5
!  q B_2/m = k1 (3 x2 x3)/r^5
!  q B_3/m = k1[(3 x3 x3)/r^5-1/r^3]
!=========================================================
subroutine F(T,Y,YP)
 include 'sr.inc'
 real(8) :: t
 real(8) :: Y(*),YP(*)
 real(8) :: x1,x2,x3,v1,v2,v3,p1,p2,p3
```

```
 real(8) :: B1,B2,B3
 real(8) :: r
 x1 = Y(1);p1 = Y(4)
 x2 = Y(2);p2 = Y(5)
 x3 = Y(3);p3 = Y(6)
 call velocity(p1,p2,p3,v1,v2,v3)
!now we can use all x1,x2,x3,p1,p2,p3,v1,v2,v3
 YP(1)  = v1
 YP(2)  = v2
 YP(3)  = v3
!Acceleration:
 r      = sqrt(x1*x1+x2*x2+x3*x3)
 if( r.gt.0.0D0)then
  B1    = k1*( 3.0D0*x1*x3)/r**5
  B2    = k1*( 3.0D0*x2*x3)/r**5
  B3    = k1*((3.0D0*x3*x3)/r**5-1/r**3)
  YP(4) = v2*B3-v3*B2
  YP(5) = v3*B1-v1*B3
  YP(6) = v1*B2-v2*B1
 else
  YP(4) = 0.0D0
  YP(5) = 0.0D0
  YP(6) = 0.0D0
 endif
end subroutine F
!=========================================================
!Energy per unit rest mass
!=========================================================
real(8) function energy(T,Y)
 include 'rk3.inc'
 real(8) :: t,e
 real(8) :: Y(*)
 real(8) :: x1,x2,x3,v1,v2,v3,p1,p2,p3,psq
 x1 = Y(1);p1 = Y(4)
 x2 = Y(2);p2 = Y(5)
 x3 = Y(3);p3 = Y(6)
 psq= p1*p1+p2*p2+p3*p3
!Kinetic Energy/m_0
 e = sqrt(1.0D0+psq)-1.0D0
 energy = e
end function energy
```

The results are shown in figure 6.8. The parameters have been exaggerated in order to achieve an aesthetically pleasant result. In reality, the electrons are moving in very thin spirals and the reader is encouraged to use more realistic values for the parameters $\vec{v}_0$, $B_0$, $\vec{r}_0$. The problem of why the effect is not seen near the equator is left as an exercise.

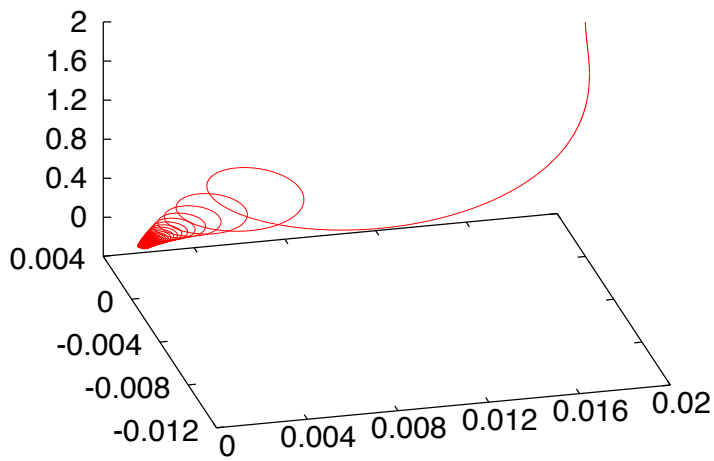



Figure 6.8: The trajectory of a charged particle in a magnetic dipole field given by equation (6.11). We used $B_0 = 1000$, $\vec{r} = 0.02\hat{x} + 2.00\hat{z}$, $\vec{v} = -0.99999\hat{z}$. The integration was done from $t_0 = 0$ to $t_f = 5$ in 10000 steps.

## 6.4 Problems

6.1 Compute the trajectory of a projectile moving in space in a constant gravitational field and under the influence of an air resistance proportional to the square of its speed.

6.2 Two point charges are moving with non relativistic speeds in a constant magnetic field $\vec{B} = B\hat{z}$. Assume that their interaction is given by the Coulomb force only. Write a program that computes their trajectory numerically using the RK45 method.

6.3 Write a program that computes the trajectory of the anisotropic harmonic oscillator $\vec{F} = -k_x x\hat{x} - k_y y\hat{y} - k_z z\hat{z}$. Compute the three dimensional Lissajous curves which appear for appropriate values of the angular frequencies $\omega_x = \sqrt{k_x/m}$, $\omega_y = \sqrt{k_y/m}$, $\omega_z = \sqrt{k_z/m}$.

6.4 Two particles of mass $M$ are at the fixed positions $\vec{r}_1 = a\hat{z}$ and $\vec{r}_2 = -a\hat{z}$. A third particle of mass $m$ interacts with them via a Newtonian gravitational force and moves at non relativistic speeds.

Compute the particle's trajectory and find initial conditions that result in a planar motion.

6.5 Solve problem 5.19 of page 283 using the RK45 method. Choose initial conditions so that the system executes only translational motion. Next, choose initial conditions so that the system executes small vibrations and its center of mass remains stationary. Find the normal modes of the system and choose appropriate initial conditions that put the system in each one of them.

6.6 Solve the previous problem by putting the system in a box $|x| \leq L$ and $|y| \leq L$.
Hint: Look in the file `springL.f90`.

6.7 Solve the problem 5.20 in page 284 by using the RK45 method.

6.8 Solve the problem 5.21 in page 284 by using the RK45 method.

6.9 The electric field of an electric dipole $\vec{p} = p\hat{z}$ is given by:

$$\begin{aligned} \vec{E} &= E_\rho \hat{\rho} + E_z \hat{z} \\ E_\rho &= \frac{1}{4\pi\epsilon_0} \frac{3p \sin\theta \cos\theta}{r^3} \\ E_z &= \frac{1}{4\pi\epsilon_0} \frac{p(3\cos^2\theta - 1)}{r^3} \end{aligned} \tag{6.12}$$

where $\rho = \sqrt{x^2 + y^2} = r \sin\theta$, $E_x = E_\rho \cos\phi$, $E_y = E_\rho \sin\phi$ and $(r, \theta, \phi)$ are the polar coordinates of the point where the electric field is calculated. Calculate the trajectory of a test charge moving in this field at non relativistic speeds. Calculate the deviation between the relativistic and the non relativistic trajectories when the initial speed is $0.01c, 0.1c, 0.5c, 0.9c$ respectively (ignore radiation effects).

6.10 Consider a linear charge distribution with constant linear charge density $\lambda$. The electric field is given by

$$\vec{E} = E_\rho \hat{\rho} = \frac{1}{4\pi\epsilon_0} \frac{2\lambda}{\rho} \hat{\rho}$$

Calculate the trajectories of two equal negative test charges that move at non relativistic speeds in this field. Consider only the electrostatic Coulomb forces and ignore anything else.

6.11 Consider a linear charge distribution on four straight lines parallel to the $z$ axis. The linear charge density is $\lambda$ and it is constant. The four straight lines intersect the $xy$ plane at the points $(0,0)$, $(0,a)$, $(a,0)$, $(a,a)$. Calculate the trajectory of a non relativistic charge in this field. Next, compute the relativistic trajectories (ignore all radiation effects).

6.12 Three particles of mass $m$ interact via their Newtonian gravitational force. Compute their (non relativistic) trajectories in space.

# Chapter 7

# Electrostatics

In this chapter we will study the electric field generated by a static charge distribution. First we will compute the electric field lines and the equipotential surfaces of the electric field generated by a static point charge distribution on the plane. Then we will study the electric field generated by a continuous charge distribution on the plane. This requires the numerical solution of an *elliptic* boundary value problem which will be done using successive over-relaxation (SOR) methods.

## 7.1 Electrostatic Field of Point Charges

Consider $N$ point charges $Q_i$ which are located at fixed positions on the plane given by their position vectors $\vec{r}_i$, $i = 1, \ldots, N$. The electric field is given by Coulomb's law

$$\vec{E}(\vec{r}) = \frac{1}{4\pi\epsilon_0} \sum_{i=1}^{N} \frac{Q_i}{|\vec{r} - \vec{r}_i|^2} \, \hat{\rho}_i \tag{7.1}$$

where $\hat{\rho}_i = (\vec{r} - \vec{r}_i)/|\vec{r} - \vec{r}_i|$ is the unit vector in the direction of $\vec{r} - \vec{r}_i$. The components of the field are

$$\begin{aligned} E_x(x,y) &= \frac{1}{4\pi\epsilon_0} \sum_{i=1}^{N} \frac{Q_i(x - x_i)}{\left((x - x_i)^2 + (y - y_i)^2\right)^{3/2}} \\ E_y(x,y) &= \frac{1}{4\pi\epsilon_0} \sum_{i=1}^{N} \frac{Q_i(y - y_i)}{\left((x - x_i)^2 + (y - y_i)^2\right)^{3/2}} \, , \end{aligned} \tag{7.2}$$

The electrostatic potential at $\vec{r}$ is

$$V(\vec{r}) = V(x,y) = \frac{1}{4\pi\epsilon_0} \sum_{i=1}^{N} \frac{Q_i}{((x-x_i)^2 + (y-y_i)^2)^{1/2}}\,, \tag{7.3}$$

and we have that

$$\vec{E}(\vec{r}) = -\vec{\nabla} V(\vec{r})\,. \tag{7.4}$$

The electric field lines are the integral curves of the vector field $\vec{E}$, i.e. the curves whose tangent lines at each point are parallel to the electric field at that point. The magnitude of the electric field is proportional to the density of the field lines (the number of field lines per perpendicular area). This means that the electric flux $\Phi_E = \int_{\mathcal{S}} \vec{E} \cdot d\vec{A}$ through a surface $\mathcal{S}$ is proportional to the number of field lines that cross the surface. Electric field lines of point charge distributions start from positive charges (sources), end in negative charges (sinks) or extend to infinity.

The equipotential surfaces are the loci of the points of space where the electrostatic potential takes fixed values. They are closed surfaces. Equation (7.4) tells us that a strong electric field at a point is equivalent to a strong spatial variation of the electric potential at this point, i.e. to dense equipotential surfaces. The direction of the electric field is perpendicular to the equipotential surfaces at each point[4], which is the direction of the strongest spatial variation of $V$, and it points in the direction of decreasing $V$. The planar cross sections of the equipotential surfaces are closed curves which are called equipotential lines.

The computer cannot solve a problem in the continuum and we have to consider a finite discretization of a field line. A continuous curve is approximated by a large but finite number of small line segments. The basic idea is illustrated in figure 7.1: The small line segment $\Delta l$ is taken in the direction of the electric field and we obtain

$$\Delta x = \Delta l\ \frac{E_x}{E}\,, \qquad \Delta y = \Delta l\ \frac{E_y}{E}\,, \tag{7.5}$$

where $E \equiv |\vec{E}| = \sqrt{E_x^2 + E_y^2}$.

In order to calculate the equipotential lines we use the property that they are perpendicular to the electric field at each point. Therefore, if $(\Delta x, \Delta y)$ is in the tangential direction of a field line, then $(-\Delta y, \Delta x)$ is

[4]Since for every small displacement $d\vec{r}$ along an equipotential surface the potential stays constant ($dV = 0$), we have that $0 = dV = \vec{\nabla} V \cdot d\vec{r} = -\vec{E} \cdot d\vec{r}$, which implies $\vec{E} \perp d\vec{r}$.

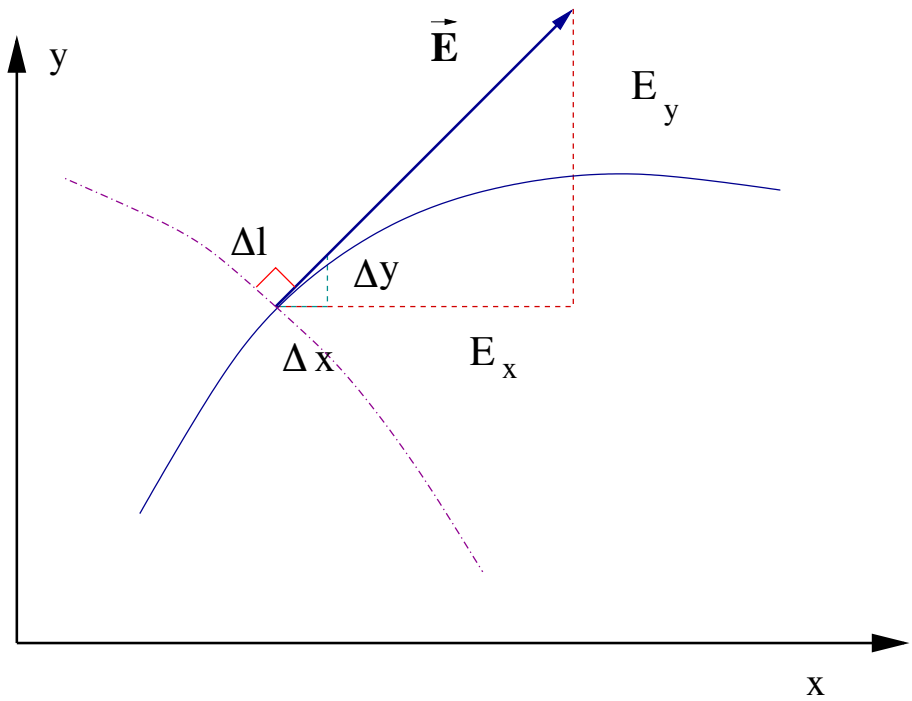



Figure 7.1: The electric field is tangent at each point of an electric field line and perpendicular to an equipotential line. By approximating the continuous curve by the line segment $\Delta l$, we have that $\Delta y/\Delta x = E_y/E_x$.

in the perpendicular direction since $(\Delta x, \Delta y) \cdot (-\Delta y, \Delta x) = -\Delta x \Delta y + \Delta y \Delta x = 0$. Therefore the equations that give the equipotential lines are

$$\Delta x = -\Delta l \, \frac{E_y}{E} \,, \qquad \Delta y = \Delta l \, \frac{E_x}{E} \,. \tag{7.6}$$

The algorithm that will allow us to perform an approximate calculation of the electric field lines and the equipotential lines is the following: Choose an initial point that belongs to the (unique) line that you want to draw. The electric field can be calculated from the known electric charge distribution and equation (7.2). By using a small enough step $\Delta l$ we move in the direction $(\Delta x, \Delta y)$ to the new position

$$x \to x + \Delta x \,, \quad y \to y + \Delta y \,, \tag{7.7}$$

where we use equations (7.5) or (7.6). The procedure is repeated until the drawing is finished. The programmer sets a criterion for that, e.g. when the field line steps out of the drawing area or approaches a charge closer than a minimum distance.

## 7.2 The Program – Appetizer and ... Desert

The hurried, but slightly experienced reader may skip the details of this section and go directly to section 7.4. There she can find the final form of the program and brief usage instructions.

In order to program the algorithm described in the previous section, we will separate the algorithmic procedures into four different but well defined tasks:

- Main program: The data structure, which is given by the position of the charges stored in the arrays `X(P)`, `Y(P)` and the charges stored in the array `Q(P)`, is defined. It also contains the user interface which consists of reading data entered by the user, like the number of charges `N`, their positions and magnitude. Then the calculation of a group of field or equipotential lines is performed by calling the routines `eline` or `epotline` respectively.

- `subroutine eline(xin,yin,X,Y,Q,N)`: Calculates the electric field line passing through the point `xin,yin`. On entry, the user inputs the point `xin,yin` and the data `N, X(N), Y(N), Q(N)`. On exit, the subroutine prints to the `stdout` the coordinates of the approximate electric field line. The line extends up to a point that is either too close to one of the point charges or until the line leaves the drawing area[2]. It calls the subroutines `efield` for the calculation of the electric field and `mdist` for the calculation of the minimum and maximum distance of a point on the field line from all the point charges.

- `subroutine epotline(xin,yin,X,Y,Q,N)`: Calculates the equipotential line passing through the point `xin,yin`. On entry, the user inputs the point `xin,yin` and the data `N, X(N), Y(N), Q(N)`. On exit, the subroutine prints to the `stdout` the coordinates of the approximate equipotential line. The subroutine stops calculating an equipotential line when it comes back close enough to the original point[3] `xin,yin` or when it leaves the drawing area. It calls the subroutines `efield` for the calculation of the electric field and `mdist` for the calculation of the minimum and maximum distance of a point on the equipotential line from all the point charges.

[2]Remember that field lines start at sources, end at sinks or extend to infinity.
[3]Remember that the equipotential lines are closed.

- `subroutine efield(x0,y0,X,Y,Q,N,Ex,Ey)`: Calculates the electric field `Ex, Ey` at position `x0, y0`. On entry, the user provides the number of charges `N`, the position of charges `X(N)`, `Y(N)`, the charges `Q(N)` and the position `x0, y0`. On exit, the routine provides the values `Ex, Ey`.

- `subroutine mdist(x0,y0,X,Y,N,rmin,rmax)`: Calculates the maximum and minimum distance of the point `x0, y0` from all charges located at `X(N), Y(N)`. On entry, the user provides the number of charges `N`, the position of charges `X(N), Y(N)` and the point `x0, y0`. On exit, the routine provides the minimum and maximum distances `rmin,rmax`.

In the main program, the variables `N, X(N), Y(N)` and `Q(N)` must be set. These can be hard coded by the programmer or entered by the user interactively. The first choice is coded in the program listed below, which can be found in the file ELines.f90. This is version 1 of the main program:

```
!*****************************************************
program Electric_Fields
!*****************************************************
 implicit none
 integer,parameter :: P=20      !max number of charges
 real,dimension(P) :: X,Y,Q
 integer           :: N
!-------------- SET CHARGE DISTRIBUTION ----
 N    =  2
 X(1) =  1.0
 Y(1) =  0.0
 Q(1) =  1.0
 X(2) = -1.0
 Y(2) =  0.0
 Q(2) = -1.0
!-------------- DRAWING LINES ---------------
 call eline(0.0, 0.5,X,Y,Q,N)
 call eline(0.0, 1.0,X,Y,Q,N)
 call eline(0.0, 1.5,X,Y,Q,N)
 call eline(0.0, 2.0,X,Y,Q,N)
 call eline(0.0,-0.5,X,Y,Q,N)
 call eline(0.0,-1.0,X,Y,Q,N)
 call eline(0.0,-1.5,X,Y,Q,N)
 call eline(0.0,-2.0,X,Y,Q,N)

end program Electric_Fields
```

The commands

```
!————————————— SET CHARGE DISTRIBUTION ———
 N    =  2
 X(1) =  1.0
 Y(1) =  0.0
 Q(1) =  1.0
 X(2) = -1.0
 Y(2) =  0.0
 Q(2) = -1.0
```

define two opposite charges `Q(1)= -Q(2)= 1.0` located at $(1,0)$ and $(-1,0)$ respectively. The next lines call the subroutine `eline` in order to perform the calculation of 8 field lines passing through the points $(0,\pm 1/2)$, $(0,\pm 1)$, $(0,\pm 3/2)$, $(0,\pm 2)$:

```
!————————————— DRAWING LINES —————————————
 call eline(0.0, 0.5,X,Y,Q,N)
 call eline(0.0, 1.0,X,Y,Q,N)
 call eline(0.0, 1.5,X,Y,Q,N)
 call eline(0.0, 2.0,X,Y,Q,N)
 call eline(0.0,-0.5,X,Y,Q,N)
 call eline(0.0,-1.0,X,Y,Q,N)
 call eline(0.0,-1.5,X,Y,Q,N)
 call eline(0.0,-2.0,X,Y,Q,N)
```

These commands print the coordinates of the field lines to the `stdout` and the user can analyze them further.

The program for calculating the equipotential lines is quite similar. The calls to the subroutine `eline` are substituted by calls to `epotline`.

For the program to be complete, we must program the subroutines `eline, efield, mdist`. This will be done later, and you can find the full code in the file `ELines.f90`. For the moment, let's copy the main program[4] listed above into the file `Elines.f90` and compile and run it with the commands:

```
> gfortran ELines.f90 -o el
> ./el > el.out
```

The `stdout` of the program is redirected to the file `el.out`. We can plot the results with `gnuplot`:

[4] See the file `ELines_version0.f90`.

```
gnuplot> plot "el.out" with dots
```

The result is shown in figure 7.2.

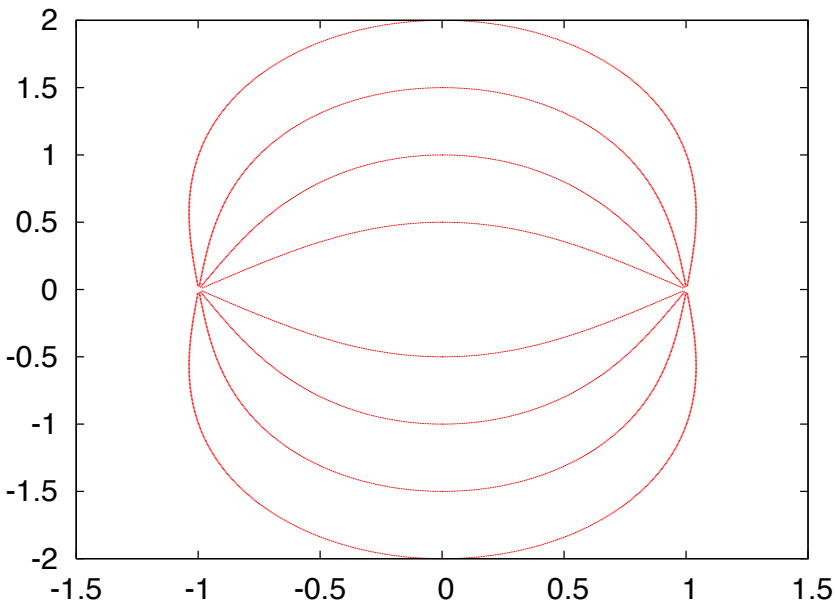



Figure 7.2: Some electric field lines of the electric field of two opposite charges calculated by the program `ELines.f90` (version 1!).

Let's modify the program so that the user can enter the charge distribution, as well as the number and position of the field lines that she wants to draw, interactively. The part of the code that we need to change is:

```
!-------------- SET CHARGE DISTRIBUTION ----
 print *, '# Enter number of charges:'
 read  *, N
 print *, '# N= ',N
 do i=1,N
  print *,'# Charge: ',i
  print *,'# Position and charge: (X,Y,Q):'
  read  *,  X(i),Y(i),Q(i)
  print *,'# (X,Y)= ', X(i),Y(i), ' Q= ',Q(i)
 enddo
```

The first line asks the user to enter the number of charges in the distribution. It proceeds with reading it from the `stdin` and prints the result

to the `stdout`. The following loop reads the positions and charges and stores them at the position `i` of the arrays `X(i)`, `Y(i)`, `Q(i)`. The results are printed to the `stdout` so that the user can check the values read by the program.

The drawing of the field lines is now done by modifying the code so that:

```
!————————————— DRAWING LINES —————————————
 print *,'# How many lines to draw? '
 read  *, draw
 do i=1,draw
  print *,'# Initial point (x0,y0): '
  read  *, x0,y0
  call  eline(x0,y0,X,Y,Q,N)
 enddo
```

As a test case, we run the program for one charge $q = 1.0$ located at the origin and we draw one field line passing through the point $(0.1, 0.1)$.

```
>  gfortran ELines.f90 -o el
> ./el
 # Enter number of charges:
1
 # N=            1
 # Charge:            1
 # Position and charge: (X,Y,Q):
0.0  0.0  1.0
 # (X,Y)=    0.000000       0.000000      Q=    1.000000
 # How many lines to draw?
1
 # Initial point (x0,y0):
0.1  0.1
  9.2928931E-02  9.2928931E-02
  8.5857861E-02  8.5857861E-02
  7.8786790E-02  7.8786790E-02
....
```

For charge distributions with a large number of point charges, use an editor to record the charges, their positions and the points where the field lines should go through.

```
2                    N: Number of Charges
 1.0  0.0   1.0      (X,Y,Q): Position and charge
-1.0  0.0  -1.0      (X,Y,Q): Position and charge
8                    Number of lines to draw
```

```
0.0   0.5              x0,y0: Initial point of line
0.0   1.0              x0,y0: Initial point of line
0.0   1.5              x0,y0: Initial point of line
0.0   2.0              x0,y0: Initial point of line
0.0  -0.5              x0,y0: Initial point of line
0.0  -1.0              x0,y0: Initial point of line
0.0  -1.5              x0,y0: Initial point of line
0.0  -2.0              x0,y0: Initial point of line
```

If the data listed above is written into a file, e.g. `Input`, then the command

```
./el < Input > el.out
```

reads the data from the file `Input` and redirects the data printed to the `stdout` to the file `el.out`. This way you can create a "library" of charge distributions and the field lines of their respective electric fields. The complete code (version 2) is listed below:

```
!*****************************************************
program Electric_Fields
!*****************************************************
 implicit none
 integer,parameter :: P=20      !max number of charges
 real,dimension(P) :: X,Y,Q
 integer           :: N
 integer           :: i,j,draw
 real              :: x0,y0
!-------------- SET CHARGE DISTRIBUTION ----
 print *, '# Enter number of charges:'
 read  *, N
 print *, '# N= ',N
 do i=1,N
  print *,'# Charge: ',i
  print *,'# Position and charge: (X,Y,Q):'
  read  *,  X(i),Y(i),Q(i)
  print *,'# (X,Y)= ', X(i),Y(i), ' Q= ',Q(i)
 enddo
!-------------- DRAWING LINES ---------------
 print *,'# How many lines to draw? '
 read  *, draw
 do i=1,draw
  print *,'# Initial point (x0,y0): '
  read  *, x0,y0
  call  eline(x0,y0,X,Y,Q,N)
 enddo
end program Electric_Fields
```

If you did the exercises described above, you should have already realized that in order to draw a nice representative picture of the electric field can be time consuming. For field lines one can use simple physical intuition in order to automate the procedure. For distances close enough to a point charge the electric field is approximately isotropic. The number of field lines crossing a small enough curve which contains only the charge is proportional to the charge (Gauss's law). Therefore we can draw a small circle centered around each charge and choose initial points isotropically distributed on the circle as initial points of the field lines. The code listed below (version 3) implements the idea for charges that are equal in magnitude. For charges different in magnitude, the program is left as an exercise to the reader.

```
!*****************************************************
program Electric_Fields
!*****************************************************
 implicit none
 integer,parameter:: P=20       !max number of charges
 real,dimension(P):: X,Y,Q
 integer           :: N
 integer           :: i,j,nd
 real              :: x0,y0,theta
 real,parameter    :: PI= 3.14159265359
!————————— SET CHARGE DISTRIBUTION ———
 print *, '# Enter number of charges:'
 read  *, N
 print *, '# N= ',N
 do i=1,N
  print *,'# Charge: ',i
  print *,'# Position and charge: (X,Y,Q):'
  read  *,  X(i),Y(i),Q(i)
  print *,'# (X,Y)= ', X(i),Y(i), ' Q= ',Q(i)
 enddo
!————————— DRAWING LINES ————————————
!We draw 2*nd field lines around each charge
 nd    = 6
 do  i = 1,N
  do j = 1,(2*nd)
   theta = (PI/nd)*j
   x0    = X(i) + 0.1 * cos(theta)
   y0    = Y(i) + 0.1 * sin(theta)
   call  eline(x0,y0,X,Y,Q,N)
  enddo
 enddo
```

```
end program Electric_Fields
```

We set the number of field lines around each charge to be equal to 12 (`nd=6`). The initial points are taken on the circle whose center is `(X(i),Y(i))` and its radius is `0.1`. The `2*nd` points are determined by the angle `theta=(PI/nd)*j`.

We record the data of a charge distribution in a file, e.g. `Input`. Below, we list the example of four equal charges $q_i = \pm 1$ located at the vertices of a square:

```
4              N: Number of charges
 1  1  -1      (X,Y,Q): Position and charge
-1  1   1      (X,Y,Q): Position and charge
 1 -1   1      (X,Y,Q): Position and charge
-1 -1  -1      (X,Y,Q): Position and charge
```

Then we give the commands:

```
> gfortran ELines.f90 -o el
> ./el < Input > el.out
> gnuplot
gnuplot> plot "el.out" with dots
```

The results are shown in figures 7.3 and 7.4. The reader should determine the charge distributions that generate those fields and reproduce the figures as an exercise.

For the computation of the equipotential lines we can work in a similar way. We will follow a quick and dirty way which will not produce an accurate picture of the electric field and choose the initial points evenly spaced on an square lattice. For a better choice see problem 5. The listed code is in the file `EPotential.f90`:

```
!*****************************************************
program Electric_Potential
!*****************************************************
 implicit none
 integer,parameter :: P=20      !max number of charges
 real,dimension(P) :: X,Y,Q
 integer           :: N
 real,parameter    :: PI= 3.14159265359
 integer           :: i,j,nd
 real              :: x0,y0,rmin,rmax,L

 print *, '# Enter number of charges:'
```

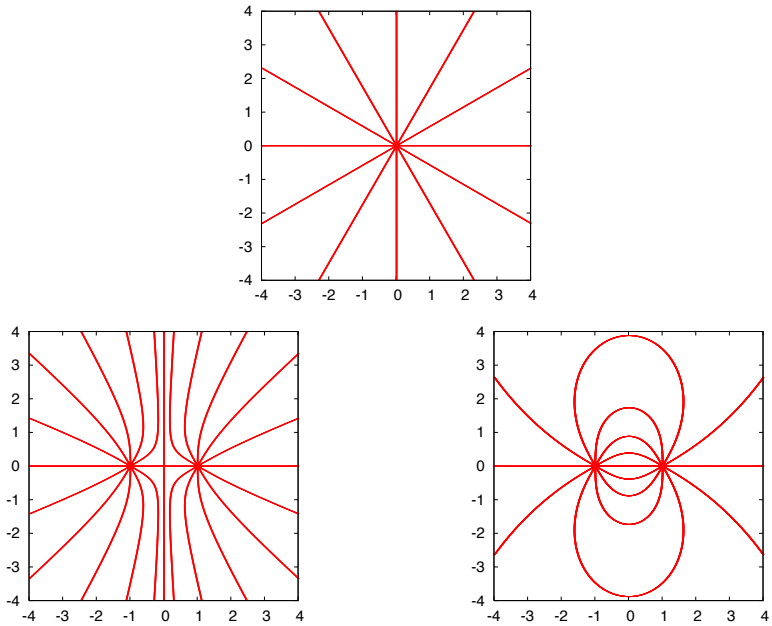



Figure 7.3: Field lines of a static charge distribution of point charges generated by the program `ELines.f90`.

```
 read   *,    N
 print *, '# N= ',N
 do i=1,N
  print *,'# Charge: ',i
  print *,'# Position and charge: (X,Y,Q):'
  read   *,  X(i),Y(i),Q(i)
  print *,'# (X,Y)= ', X(i),Y(i), ' Q= ',Q(i)
 enddo
!————————————— DRAWING LINES —————————————
!We draw lines passing through an equally
!spaced lattice of N=(2*nd+1)x(2*nd+1) points
!in the square -L<= x <= L, -L<= y <= L.
 nd     = 4
 L      = 1.0
 do  i = -nd,nd
  do j = -nd,nd
   x0  = i*(L/nd)
   y0  = j*(L/nd)
   print *,'# @ ',i,j,L/nd,x0,y0
   call  mdist(x0,y0,X,Y,N,rmin,rmax)
!we avoid getting too close to a charge:
   if(rmin .gt. L/(nd*10) )&
        call  epotline(x0,y0,X,Y,Q,N)
```

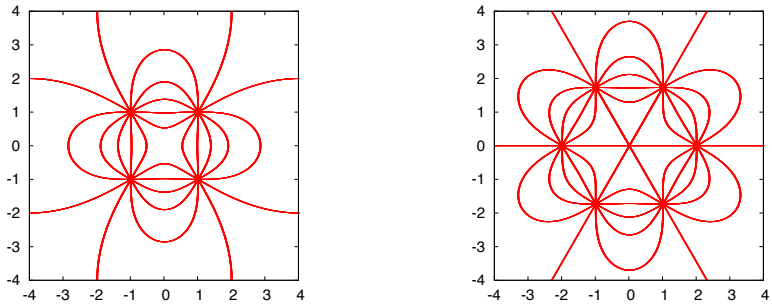



Figure 7.4: Field lines of a static charge distribution of point charges generated by the program `ELines.f90`.

```
  enddo
 enddo
end program Electric_Potential
```

The first and second part of the code is identical to the previous one. In the third part we call the subroutine `epotline` for drawing an equipotential line for each initial point. The initial points are on a square lattice with `(2*nd+1)*(2*nd+1)=` 25 points (`nd=4`). The lattice extends within the limits set by the square $(1,1)$, $(-1,1)$, $(-1,-1)$, $(1,-1)$ (`L=1.0`). For each point `(x0,y0)` we calculate the equipotential line that passes through it. We check that this point is not too close to one of the charges by calling the subroutine `mdist`. The call determines the minimum distance `rmin` of the point from all the charges which should be larger than `L/(nd*10)`. You can run the program with the commands:

```
> gfortran EPotential.f90 -o ep
> ./ep < Input > ep.out
> gnuplot
gnuplot> plot "ep.out" with dots
```

Some of the results are shown in figure 7.5.

## 7.3 The Program – Main Dish

In this section we look under the hood and give the details of the inner parts of the program: The subroutines `eline` and `epotline` that calculate the field and equipotential lines, the subroutine `efield` that calculates the electric field at a point and the subroutine `mdist` that calculates the

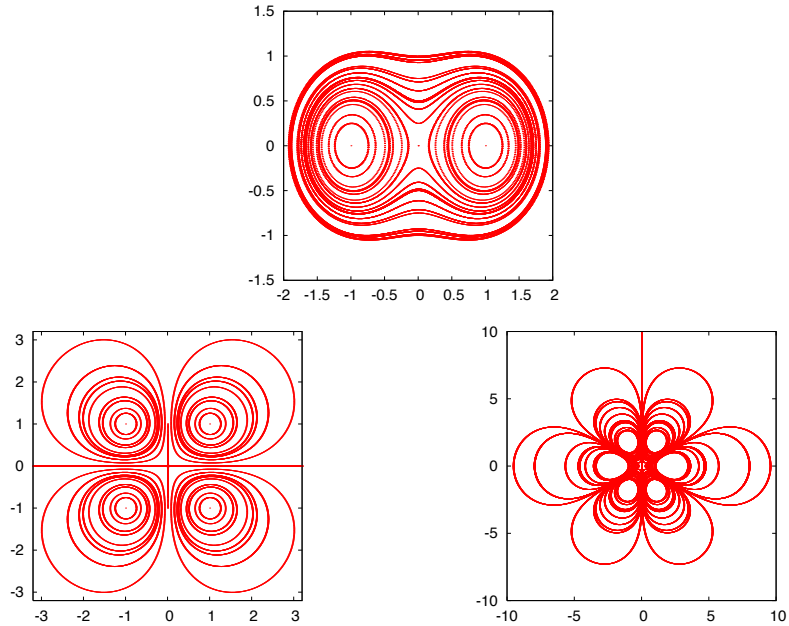



Figure 7.5: Equipotential lines of the electric field generated by a point charge distribution on the plane calculated by the program in `EPotential.f90`. Beware: the density of the lines is not correctly calculated and it is not proportional to the magnitude of the electric field. See problem 7.5.

minimum and maximum distances of a point from the point charges.

The subroutine `eline` is called by the command:

```
      call  eline(x0,y0,X,Y,Q,N)
```

The input to the routine is the initial point `(x0,y0)`, the number of charges `N`, the positions of the charges `(X(N),Y(N))` and the charges `Q(N)`. The routine needs some parameters in order to draw the field line. These are "hard coded", i.e. set to fixed values by the programmer that cannot be changed by the user that calls the routine in her program. One of them is the step $\Delta l$ of equation (7.5) which sets the discretization step of the field line. It also sets the minimum distance of approaching to a charge equal to $2\Delta l$. The size of the drawing area of the curve is set by the parameter `max_dist=20.0`. We should also provide a check in the program that checks whether the electric field is zero, in which case the result of the calculation in equation (7.5) becomes indeterminate. By taking $\Delta l > 0$, the motion is in the direction of the electric field, which ends on a negative charge or outside the drawing area (why?). In order

to draw the line in both directions, set $\Delta l < 0$ and repeat the calculation.

The code is listed below:

```
!*****************************************************
subroutine eline(xin,yin,X,Y,Q,N)
!*****************************************************
 implicit  none
 integer              :: N
 real,dimension(N):: X,Y,Q
 real                 :: xin,yin,x0,y0
 real,parameter       :: step=0.01
 real,parameter       :: max_dist=20.0
 integer              :: i,direction
 real                 :: rmin,rmax,r,dx,dy,dl
 real                 :: Ex,Ey,E
 do direction = -1,1,2                !direction= +/- 1
  dl = direction * step
  x0 = xin
  y0 = yin
  dx = 0.0
  dy = 0.0
  call mdist(x0,y0,X,Y,N,rmin,rmax)
  do while(rmin .gt. (2.0*step) .and. rmax .lt. max_dist)
   print *,x0,y0
!  We evaluate the E-field at the midpoint: This reduces
!  systematic errors
   call efield(x0+0.5*dx,y0+0.5*dy,X,Y,Q,N,Ex,Ey)
   E  = sqrt(Ex*Ex+Ey*Ey)
   if( E .le. 1.0e-10 ) exit
   dx = dl*Ex/E
   dy = dl*Ey/E
   x0 = x0 + dx
   y0 = y0 + dy
   call mdist(x0,y0,X,Y,N,rmin,rmax)
  enddo                       !do while()
 enddo                        !do direction = -1,1,2
end subroutine eline
```

In the first part of the code we have the variable declarations. We only note the declaration

```
 real,dimension(N):: X,Y,Q
```

which declares the dimension of the arrays to be `N` instead of their true dimension `P`. This is fine, as long as the programmer of the calling program has already checked that `N` $\leq$ `P`. The necessary memory for the

arrays is allocated in the calling program and the declaration does not provide new storage space. The arrays `X,Y, Q` are passed to the subroutine "by reference", i.e. the routine learns about their position in the memory to which it can refer to, and not "by value". The parameters $\Delta l =$ `step` and `max_dist` are fixed by the `parameter` attribute:

```
real,parameter    :: step=0.01
real,parameter    :: max_dist=20.0
```

Their values should be the result of a careful study by the programmer since they determine the accuracy of the calculation.

The outmost loop

```
do direction = -1,1,2
 dl = direction * step
  ...
enddo
```

sets the direction of motion on the field line (i.e. the sign of $\Delta l$). The command `do direction = -1,1,2` executes the loop twice by setting the variable `direction` to take values from $-1$ to $1$ with step equal to $2$.

The commands `x0 = xin, y0 = yin` define the initial point on the field line. `(x0, y0)` is the current point on the field line which is printed to the `stdout` with the command `print`. The variables `(dx, dy)` set the step `(x0, y0)` $\rightarrow$ `(x0+dx, y0+dy)`. The drawing of the field line is done in the inner loop

```
call mdist(x0,y0,X,Y,N,rmin,rmax)
do while(rmin .gt. (2.0*step) .and. rmax .lt. max_dist)
 ...
 call mdist(x0,y0,X,Y,N,rmin,rmax)
enddo
```

which is executed provided that the logical expression `(rmin .gt. (2.0*step) .and. rmax .lt. max_dist)` is `.TRUE`. This happens as long as the current point is at a distance greater than `2.0*step` and the maximum distance from all charges is less than max_dist[5]. The minimum and maximum distances are calculated by calling the subroutine `mdist`.

The electric field, needed in equation (7.5), is calculated by a call to `efield(x0+0.5*dx,y0+0.5*dy,X,Y,Q,N,Ex,Ey)`. The first two arguments give the point at which we want to calculate the electric field, which is

[5]The choice is not unique of course, you may also try e.g. `rmin .lt. max_dist`.

chosen to be the *midpoint* (x0+dx/2,y0+dy/2) instead of (x0,y0). This improves the stability and the accuracy of the algorithm.

Equation (7.5) is coded in the commands

```
   E  = sqrt(Ex*Ex+Ey*Ey)
   dx = dl*Ex/E
   dy = dl*Ey/E
   x0 = x0 + dx
   y0 = y0 + dy
```

We also perform checks for the cases E=0.0 and dx=dy=0.0:

```
   if( E .le. 1.0e-10 ) exit
```

When the magnitude of the electric field becomes too small we stop the calculation by exiting the loop with the command exit. The reader can improve the code by adding more checks of singular cases.

The subroutine epotline is programmed in a similar way. The relevant code is listed below:

```
!*****************************************************
subroutine epotline(xin,yin,X,Y,Q,N)
!*****************************************************
 implicit none
 integer            :: N
 real,dimension(N)  :: X,Y,Q
 real               :: xin,yin,x0,y0
 real,parameter     :: step=0.02
 real,parameter     :: max_dist=20.0
 integer            :: i
 real               :: r,dx,dy,dl
 real               :: Ex,Ey,E

 dl = step
 x0 = xin
 y0 = yin
 dx = 0.0
 dy = 0.0
 r  = step                  !in order to start loop
 do while( r .gt. (0.9*dl) .and. r .lt. max_dist)
  print *,x0,y0
! We evaluate the E-field at the midpoint: This reduces
! systematic errors
  call efield(x0+0.5*dx,y0+0.5*dy,X,Y,Q,N,Ex,Ey)
  E  = sqrt(Ex*Ex+Ey*Ey)
```

```
  if( E .le. 1.0e-10 ) exit
  dx =  dl*Ey/E
  dy = -dl*Ex/E
  x0 = x0 + dx
  y0 = y0 + dy
  r  = sqrt((x0-xin)**2+(y0-yin)**2)
 enddo                          !do while()
end subroutine epotline
```

The differences are minor: The equipotential lines are closed curves, therefore we only need to transverse them in one direction. The criterion for ending the calculation is to approach the initial point close enough or leave the drawing area:

```
 do while( r .gt. (0.9*dl) .and. r .lt. max_dist)
  ...
 enddo
```

The values of dx, dy are calculated according to equation (7.6):

```
  dx =  dl*Ey/E
  dy = -dl*Ex/E
```

The subroutine efield is an application of equations[6] (7.2):

```
!*****************************************************
subroutine efield(x0,y0,X,Y,Q,N,Ex,Ey)
!*****************************************************
 implicit none
 integer              :: N
 real,dimension(N)    :: X,Y,Q
 real                 :: x0,y0,dx,dy,Ex,Ey
 integer              :: i
 real                 :: r3,xi,yi

 Ex  = 0.0
 Ey  = 0.0
 do i= 1,N
  xi = x0-X(i)
  yi = y0-Y(i)
  r3 = (xi*xi+yi*yi)**(-1.5)
  Ex = Ex + Q(i)*xi*r3
  Ey = Ey + Q(i)*yi*r3
 enddo
```

[6]You may improve the program by checking whether $r_i = 0$.

```
end subroutine efield
```

Finally, the subroutine `mdist` calculates the minimum and maximum distance `rmin` and `rmax` of a point `(x0,y0)` from all the point charges in the distribution:

```
!*****************************************************
subroutine mdist(x0,y0,X,Y,N,rmin,rmax)
!*****************************************************
 implicit none
 integer              :: N
 real,dimension(N)    :: X,Y
 real                 :: x0,y0,rmin,rmax
 integer              :: i
 real                 :: r

 rmax = 0.0
 rmin = 1000.0
 do i = 1,N
  r  = sqrt((x0-X(i))**2 + (y0-Y(i))**2)
  if(r.GT.rmax) rmax = r
  if(r.LT.rmin) rmin = r
 enddo
end subroutine mdist
```

The initial value of `rmin` depends of the limits of the drawing area (why?).

## 7.4 The Program - Conclusion

In this section we list the programs discussed in the previous sections and provide short usage information for compiling, running and analyzing your results. You can jump into this section without reading the previous ones and go back to them if you need to clarify some points that you find hard to understand.

First we list the contents of the file `ELines.f90`:

```
!*****************************************************
program Electric_Fields
!*****************************************************
 implicit none
 integer,parameter:: P=20       !max number of charges
 real,dimension(P):: X,Y,Q
 integer          :: N
 integer          :: i,j,nd
```

```
 real                :: x0,y0,theta
 real,parameter      :: PI= 3.14159265359
!-------------- SET CHARGE DISTRIBUTION ----
 print *, '# Enter number of charges:'
 read  *, N
 print *, '# N= ',N
 do i=1,N
  print *,'# Charge: ',i
  print *,'# Position and charge: (X,Y,Q):'
  read  *,  X(i),Y(i),Q(i)
  print *,'# (X,Y)= ', X(i),Y(i), ' Q= ',Q(i)
 enddo
!-------------- DRAWING LINES --------------
!We draw 2*nd field lines around each charge
 nd    = 6
 do  i = 1,N
  do j = 1,(2*nd)
   theta = (PI/nd)*j
   x0    = X(i) + 0.1 * cos(theta)
   y0    = Y(i) + 0.1 * sin(theta)
   call  eline(x0,y0,X,Y,Q,N)
  enddo
 enddo
end program Electric_Fields
!*****************************************************
subroutine eline(xin,yin,X,Y,Q,N)
!*****************************************************
 implicit  none
 integer             :: N
 real,dimension(N):: X,Y,Q
 real                :: xin,yin,x0,y0
 real,parameter      :: step=0.01
 real,parameter      :: max_dist=20.0
 integer             :: i,direction
 real                :: rmin,rmax,r,dx,dy,dl
 real                :: Ex,Ey,E
 do direction = -1,1,2              !direction= +/- 1
  dl = direction * step
  x0 = xin
  y0 = yin
  dx = 0.0
  dy = 0.0
  call mdist(x0,y0,X,Y,N,rmin,rmax)
  do while(rmin .gt. (2.0*step) .and. rmax .lt. max_dist)
   print *,x0,y0
!  We evaluate the E-field at the midpoint: This reduces
!  systematic errors
   call efield(x0+0.5*dx,y0+0.5*dy,X,Y,Q,N,Ex,Ey)
   E  = sqrt(Ex*Ex+Ey*Ey)
```

```
    if( E .le. 1.0e-10 ) exit
    dx = dl*Ex/E
    dy = dl*Ey/E
    x0 = x0 + dx
    y0 = y0 + dy
    call mdist(x0,y0,X,Y,N,rmin,rmax)
   enddo                        !do while()
  enddo                         !do direction = -1,1,2
end subroutine eline
!*****************************************************
subroutine efield(x0,y0,X,Y,Q,N,Ex,Ey)
!*****************************************************
 implicit none
 integer              :: N
 real,dimension(N)    :: X,Y,Q
 real                 :: x0,y0,dx,dy,Ex,Ey
 integer              :: i
 real                 :: r3,xi,yi

 Ex  = 0.0
 Ey  = 0.0
 do i= 1,N
  xi = x0-X(i)
  yi = y0-Y(i)
! Exercise: Improve code so that xi*xi+yi*yi=0 is taken care of
  r3 = (xi*xi+yi*yi)**(-1.5)
  Ex = Ex + Q(i)*xi*r3
  Ey = Ey + Q(i)*yi*r3
 enddo
end subroutine efield
!*****************************************************
subroutine mdist(x0,y0,X,Y,N,rmin,rmax)
!*****************************************************
 implicit none
 integer              :: N
 real,dimension(N)    :: X,Y
 real                 :: x0,y0,rmin,rmax
 integer              :: i
 real                 :: r

 rmax = 0.0
 rmin = 1000.0
 do i = 1,N
  r  = sqrt((x0-X(i))**2 + (y0-Y(i))**2)
  if(r.GT.rmax) rmax = r
  if(r.LT.rmin) rmin = r
 enddo
end subroutine mdist
```

Then we list the contents of the file `EPotential.f90`:

```
!*****************************************************
program Electric_Potential
!*****************************************************
 implicit none
 integer,parameter :: P=20      !max number of charges
 real,dimension(P) :: X,Y,Q
 integer           :: N
 real,parameter    :: PI= 3.14159265359
 integer           :: i,j,nd
 real              :: x0,y0,rmin,rmax,L

 print *, '# Enter number of charges:'
 read  *,   N
 print *, '# N= ',N
 do i=1,N
  print *,'# Charge: ',i
  print *,'# Position and charge: (X,Y,Q):'
  read  *,  X(i),Y(i),Q(i)
  print *,'# (X,Y)= ', X(i),Y(i), ' Q= ',Q(i)
 enddo
!————————— DRAWING LINES —————————
!We draw lines passing through an equally
!spaced lattice of N=(2*nd+1)x(2*nd+1) points
!in the square -L<= x <= L, -L<= y <= L.
 nd    = 4
 L     = 1.0
 do  i = -nd,nd
  do j = -nd,nd
   x0  = i*(L/nd)
   y0  = j*(L/nd)
   print *,'# @ ',i,j,L/nd,x0,y0
   call  mdist(x0,y0,X,Y,N,rmin,rmax)
!we avoid getting too close to a charge:
   if(rmin .gt. L/(nd*10) )&
        call  epotline(x0,y0,X,Y,Q,N)
  enddo
 enddo
end program Electric_Potential
!*****************************************************
subroutine epotline(xin,yin,X,Y,Q,N)
!*****************************************************
 implicit none
 integer           :: N
 real,dimension(N) :: X,Y,Q
 real              :: xin,yin,x0,y0
 real,parameter    :: step=0.02
```

```
 real,parameter     :: max_dist=20.0
 integer            :: i
 real               :: r,dx,dy,dl
 real               :: Ex,Ey,E

 dl = step
 x0 = xin
 y0 = yin
 dx = 0.0
 dy = 0.0
 r  = step                  !in order to start loop
 do while( r .gt. (0.9*dl) .and. r .lt. max_dist)
  print *,x0,y0
! We evaluate the E-field at the midpoint: This reduces
! systematic errors
  call efield(x0+0.5*dx,y0+0.5*dy,X,Y,Q,N,Ex,Ey)
  E  = sqrt(Ex*Ex+Ey*Ey)
  if( E .le. 1.0e-10 ) exit
  dx =  dl*Ey/E
  dy = -dl*Ex/E
  x0 = x0 + dx
  y0 = y0 + dy
  r  = sqrt((x0-xin)**2+(y0-yin)**2)
 enddo                      !do while()
end subroutine epotline
...
```

where ... are the subroutines `efield` and `mdist` which are identical to the ones in the file `ELines.f90`.

In order to compile the program use the commands:

```
> gfortran ELines.f90     -o el
> gfortran EPotential.f90 -o ep
```

Then, edit a file and name it e.g. `Input` and write the data that define a charge distribution. For example:

```
4              N: Number of charges
 1  1  -1      (X,Y,Q): Position and charge
-1  1   1      (X,Y,Q): Position and charge
 1 -1   1      (X,Y,Q): Position and charge
-1 -1  -1      (X,Y,Q): Position and charge
```

The results are obtained with the commands:

```
> ./el < Input > el.dat
```

```
> ./ep < Input > ep.dat
> gnuplot
gnuplot> plot "el.dat" with dots
gnuplot> plot "ep.dat" with dots
```

Have fun!

## 7.5 Electrostatic Field in the Vacuum

Consider a time independent electric field in an area of space which is empty of electric charge. Maxwell's equations are reduced to Gauss's law

$$\vec{\nabla} \cdot \vec{E}(x, y, z) = \frac{\partial E_x}{\partial x} + \frac{\partial E_y}{\partial y} + \frac{\partial E_z}{\partial z} = 0\,, \tag{7.8}$$

together with the equation that defines the electrostatic potential[7]

$$\vec{E}(x, y, z) = -\vec{\nabla} V(x, y, z)\,. \tag{7.9}$$

Equations (7.8) and (7.9) give the Laplace equation for the function $V(x, y, z)$:

$$\nabla^2 V(x, y, z) = \frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + \frac{\partial^2 V}{\partial z^2} = 0\,. \tag{7.10}$$

The solution of the equation above is a boundary value problem: We are looking for the potential $V(x, y, z)$ in a region of space $\mathcal{S}$ bounded by a closed surface $\partial\mathcal{S}$. When the potential is known on $\partial\mathcal{S}$ the solution to (7.10) is unique and the potential and the electric field is determined everywhere in $\mathcal{S}$.

For simplicity consider the problem confined on a plane, therefore $V = V(x, y)$. In this case the last term in equation (7.10) vanishes, the region $\mathcal{S}$ is a compact subset of the plane and $\partial\mathcal{S}$ is a closed curve.

For the numerical solution of the problem, we approximate $\mathcal{S}$ by a discrete, square lattice. The potential is defined on the $N$ sites of the lattice. We take $\mathcal{S}$ to be bounded by a square with sides of length $l$. The distance between the nearest lattice points is called the lattice constant $a$. Then $l = (L - 1)a$, where $L = \sqrt{N}$ is the number of lattice points on each side of the square. The continuous solution is approximated by the solution on the lattice, and the approximation becomes exact in the $N \to \infty$ and $a \to 0$ limits, so that the length $l = (L-1)a$ remains constant. The curve $\partial\mathcal{S}$ is approximated by the lattice sites that are located on the

[7] Equivalent to the equation $\vec{\nabla} \times \vec{E} = 0$.

perimeter of the square and the loci in the square where the potential takes constant values. This is a simple model of a set of conducting surfaces (points where $V = \text{const.} \neq 0$) in a compact region whose boundary is grounded (points where $V = 0$). An example is depicted in figure 7.6.

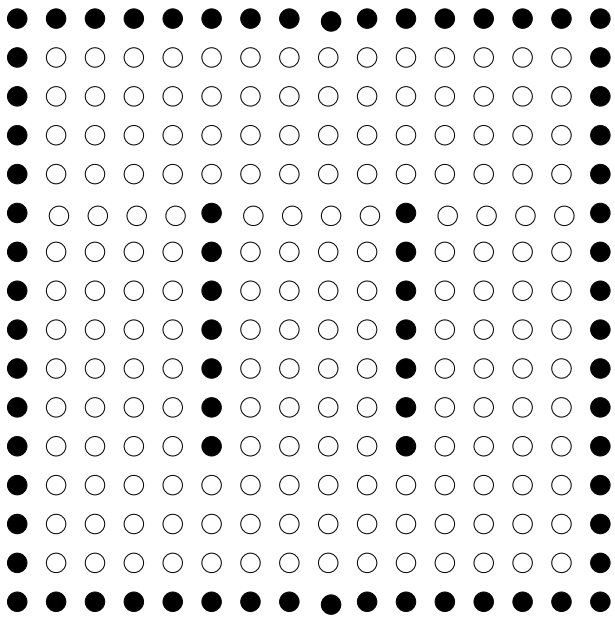


Figure 7.6: A lattice which corresponds to a cross section of two parallel conducting planes inside a grounded cubic box. The black lattice sites are the points of constant, fixed potential whereas the white ones are sites in the vacuum.

In order to derive a finite difference equation which approximates equation (7.10), we Taylor expand around a point $(x, y)$ according to the equations:

$$\begin{aligned}
V(x+\delta x, y) &= V(x,y) + \frac{\partial V}{\partial x}\delta x + \frac{1}{2}\frac{\partial^2 V}{\partial x^2}(\delta x)^2 + \dots \\
V(x-\delta x, y) &= V(x,y) - \frac{\partial V}{\partial x}\delta x + \frac{1}{2}\frac{\partial^2 V}{\partial x^2}(\delta x)^2 + \dots \\
V(x, y+\delta y) &= V(x,y) + \frac{\partial V}{\partial y}\delta y + \frac{1}{2}\frac{\partial^2 V}{\partial y^2}(\delta y)^2 + \dots \\
V(x, y-\delta y) &= V(x,y) - \frac{\partial V}{\partial y}\delta y + \frac{1}{2}\frac{\partial^2 V}{\partial y^2}(\delta y)^2 + \dots .
\end{aligned}$$

By summing both sides of the above equations, taking $\delta x = \delta y$ and

ignoring the terms implied by ..., we obtain

$$\begin{aligned} & V(x+\delta x, y) + V(x-\delta x, y) + V(x, y+\delta y) + V(x, y-\delta y) \\ & \quad = 4V(x,y) + (\delta x)^2 (\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2}) + \ldots \\ & \quad \approx 4V(x,y)\,, \end{aligned} \tag{7.11}$$

The second term in the second line was eliminated by using equation (7.10).

We map the coordinates of the lattice points to integers $(i, j)$ such that $x_i = (i-1)a$ and $y_j = (j-1)a$ where $i, j = 1, \ldots, L$. By taking $\delta x = \delta y = a$ so that $x_i \pm \delta x = x_i \pm a = (i - 1 \pm 1)a = x_{i\pm 1}$ and $y_j \pm \delta y = y_j \pm a = (j - 1 \pm 1)a = y_{j\pm 1}$, equation (7.11) becomes:

$$V(i,j) = \frac{1}{4}(V(i-1,j) + V(i+1,j) + V(i,j-1) + V(i,j+1))\,. \tag{7.12}$$

The equation above states that the potential at the position $(i, j)$ is the arithmetic mean of the potential of the nearest neighbors. We will describe an algorithm which belongs to the class of "successive overrelaxation methods" (SOR) whose basic steps are:

1. Set the size $L$ of the square lattice.

2. Flag the sites that correspond to "conductors", i.e. the sites where the potential remains fixed to the boundary conditions values.

3. Choose an initial trial function for $V(x, y)$ on the vacuum sites. Of course it is not the solution we are looking for. A good choice will lead to fast convergence of the algorithm to the true solution. A bad choice may lead to slow convergence, no convergence or even convergence to the wrong solution. In our case the problem is easy and the simple choice $V(x, y) = 0$ will do.

4. Sweep the lattice and enforce equation (7.12) on each visited vacuum site. This defines the new value of the potential at this site.

5. Sweep the lattice repeatedly until two successive sweeps result in a very small change in the function $V(x, y)$.

A careful study of the above algorithm requires to test different criteria of "very small change" and test that different choices of the initial function $V(x, y)$ result in the same solution.

We write a program that implements this algorithm in the case of a system which is the projection of two parallel conducting planes inside a grounded cubic box on the plane. The lattice is depicted in figure 7.6, where the black dots correspond to the conductors. All the points of the box have $V = 0$ and the two conductors are at constant potential $V_1$ and $V_2$ respectively. The user enters the values $V_1$ and $V_2$, the lattice size $L$ and the required accuracy interactively. The latter is determined by a small number $\epsilon$. The convergence criterion that we set is that the maximum difference between the values of the potential between two successive sweeps should be less than $\epsilon$.

The data structure is very simple. We use a real array `V(L,L)` in order to store the values of the potential at each lattice site. A logical array `isConductor(L,L)` flags each site as a "conductor site" (= `.TRUE.`) or as a "vacuum site" (=`.FALSE.`).

The main program reads in the data entered by the user and then calls three subroutines:

1. `initialize_lattice(V,isConductor,L,V1,V2)`:
   The routine needs at its input the values of the potential `V1` and `V2` on the left and right plate respectively and the size of the lattice `L`. On exit it provides the initial values of the potential `V(L,L)` and the flags `isConductor(L,L)`. The geometry of the setting is hard coded and the user needs to change this subroutine each time that she wants to study a different geometry.

2. `laplace(V,isConductor,L,epsilon)`:
   This is the heart of the program. On entry we provide the initialized arrays `V(L,L)` and `isConductor(L,L)`, the lattice size `L`, and the desired accuracy `epsilon`. On exit we obtain the final solution `V(L,L)`. This subroutine calculates the arithmetic mean of the potential of the nearest neighbors `Vav` and the value `V(i,j)=Vav` is changed immediately[8]. The maximum change in the new value of the potential `Vav` from the old one `V(i,j)` is stored in the variable `error`. When `error` becomes smaller than `epsilon` we assume that convergence has been achieved.

3. `print_results(V,L)`:
   This subroutine prints the potential `V(L,L)` to the file `data`. Each line contains the integers `i, j` and the value of the potential `V(i,j)`.

[8]A different choice would have been to store the value `Vav` in a temporary array `Vnew(i,j)`. After the sweep, the potential `V(i,j)=Vnew(i,j)` is changed to the new values. Which method do you expect to have better convergence properties? Try...

We note that each time that the index `i` changes, the subroutine prints an extra empty line. This is done so that the output can be read easily by the three dimensional plotting function `splot` of `gnuplot`.

The full program is listed below:

```
!*************************************************************
!PROGRAM LAPLACE_EM
!Computes the electrostatic potential around conductors.
!The computation is performed on a square lattice of linear
!dimension L. A relaxation method is used to converge to the
!solution of Laplace equation for the potential.
!DATA STRUCTURE:
!real(8) V(L,L): Value of the potential on the lattice sites
!logical isConductor(L,L): If .TRUE.  site has fixed potential
!                          If .FALSE. site is empty space
!real epsilon: Determines the accuracy of the solution
!The maximum difference of the potential on each site between
!two consecutive sweeps should be less than epsilon.
!PROGRAM STRUCTURE
!main program:
! . Data Input
! . call subroutines for initialization, computation and
!   printing of results
!subroutine initialize_lattice:
! . Initilization of V(L,L) and isConductor(L,L)
!subroutine laplace:
! . Solves laplace equation using a relaxation method
!subroutine print_results:
! . Prints results for V(L,L) in a file. Uses format compatible
!with splot of gnuplot.
!*************************************************************
program laplace_em
 implicit none
!P defines the size of the arrays and is equal to L
 integer,parameter       :: P=31
 logical,dimension(P,P) :: isConductor
 real(8),dimension(P,P) :: V
!V1 and V2 are the values of the potential on the interior
!conductors. epsilon is the accuracy desired for the
!convergence of the relaxation method in subroutine
!laplace()
 real(8)                 :: V1,V2,epsilon
 integer                 :: L

!We ask the user to provide the necessary data:
!V1,V2 and epsilon
```

```
 L = P
 print *,'Enter V1,V2:'
 read  *, V1,V2
 print *,'Enter epsilon:'
 read  *, epsilon
 print *,'Starting Laplace:'
 print *,'Grid Size= ',L
 print *,'Conductors set at V1= ',V1,' V2= ',V2
 print *,'Relaxing with accuracy epsilon= ',epsilon
!The arrays V and isConductor are initialized
 call initialize_lattice(V,isConductor,L,V1,V2)
!We enter initialized V,isConductor. On exit the
!routine gives the solution V
 call laplace(V,isConductor,L,epsilon)
!We print V in a file.
 call print_results(V,L)

end program laplace_em
!*************************************************************
!subroutine initialize_lattice
!Initializes arrays V(L,L) and isConductor(L,L).
!V(L,L)= 0.0  and isConductor(L,L)= .FALSE. by default
!isConductor(i,j)= .TRUE. on boundary of lattice where V=0
!isConductor(i,j)= .TRUE. on sites with i=  L/3+1, 5<= j <= L-5
!isConductor(i,j)= .TRUE. on sites with i=2*L/3+1, 5<= j <= L-5
!V(i,j) = V1  on all sites with i=  L/3+1, 5<= j <= L-5
!V(i,j) = V2  on all sites with i=2*L/3+1, 5<= j <= L-5
!V(i,j) = 0   on boundary (i=1,L and j=1,L)
!V(i,j) = 0   on interior sites with isConductor(i,j)= .FALSE.
!INPUT:
!integer L: Linear size of lattice
!real(8)  V1,V2: Values of potential on interior conductors
!OUTPUT:
!real(8)  V(L,L): Array provided by user. Values of potential
!logical isConductor(L,L): If .TRUE.  site has fixed potential
!                          If .FALSE. site is empty space
!*************************************************************
subroutine initialize_lattice(V,isConductor,L,V1,V2)
 implicit none
 integer                 :: L
 logical,dimension(L,L)  :: isConductor
 real(8),dimension(L,L)  :: V
 real(8)                 :: V1,V2
 integer                 :: i,j

!Initialize to 0 and .FALSE (default values for boundary and
!interior sites).
 V           =  0.0D0
 isConductor = .FALSE.
```

```
!We set the boundary to be a conductor: (V=0 by default)
 do i=1,L
  isConductor(1,i) = .TRUE.
  isConductor(i,1) = .TRUE.
  isConductor(L,i) = .TRUE.
  isConductor(i,L) = .TRUE.
 enddo
!We set two conductors at given potential V1 and V2
 do i=5,L-5
  V           (  L/3+1,i) =  V1
  isConductor(  L/3+1,i) = .TRUE.
  V           (2*L/3+1,i) =  V2
  isConductor(2*L/3+1,i) = .TRUE.
 enddo

end subroutine initialize_lattice
!*************************************************************
!subroutine laplace
!Uses a relaxation method to compute the solution of the
!Laplace equation for the electrostatic potential
!on a 2 dimensionalsquarelattice of linear size L.
!At every sweep of the lattice we compute the average
!Vav of thepotential at each site (i,j) and we immediately
! update V(i,j).
!The computation continues until Max |Vav-V(i,j)| < epsilon
!INPUT:
!integer L: Linear size of lattice
!real(8) V(L,L): Value of the potential at each site
!logical isConductor(L,L): If .TRUE.  potential is fixed
!                          If .FALSE. potential is updated
!real(8) epsilon: if Max |Vav-V(i,j)| < epsilon return to
!callingprogram.
!OUTPUT:
!real(8) V(L,L): The computed solution for the potential
!*************************************************************
subroutine laplace(V,isConductor,L,epsilon)
 implicit none
 integer :: L
 logical,dimension(L,L) :: isConductor
 real(8),dimension(L,L) :: V
 real(8)                :: epsilon
 integer                :: i,j,icount
 real(8)                :: Vav,error,dV

 icount = 0                 !counts number of sweeps
 do while (.TRUE.)          !an infinite loop:
  error = 0.0D0             !Exit when error<epsilon
  do  j=2,L-1
   do i=2,L-1
```

```
!We change V only for non conductors:
    if( .NOT. isConductor(i,j))then
     Vav = ( V(i-1,j)+V(i+1,j)+V(i,j+1)+V(i,j-1)) * 0.25D0
     dV  = DABS(V(i,j)-Vav)
     if(error .LT. dV ) error = dV !maximum error
     V(i,j) = Vav          ! we immendiately update V(i,j)
    endif
   enddo
  enddo
  icount = icount + 1
  print *,icount,' err= ',error
  if( error .LT. epsilon) return !return to main program
 enddo

end subroutine laplace
!*************************************************************
!subroutine  print_results
!Prints the array V(L,L) in file "data"
!The format of the output is appropriate for the splot function
!of gnuplot: Each time i changes an empty line is printed.
!INPUT:
!integer L: size of array V
!real(8) V(L,L): array to be printed
!OUTPUT:
!no output
!*************************************************************
subroutine  print_results(V,L)
 implicit none
 integer                 :: L
 real(8),dimension(L,L) :: V
 integer                 :: i,j

 open(unit=11,file="data")
 do i=1,L
  do j =1,L
   write(11,*)i,j,V(i,j)
  enddo
  write (11,*)'' !empty line for gnuplot,separate isolines
 enddo

end subroutine print_results
```

## 7.6 Results

The program in the previous section is written in the file `LaplaceEq.f90`. Compiling and running is done with the commands:

```
> gfortran LaplaceEq.f90 -o lf
> ./lf
 Enter V1,V2:
100 -100
 Enter epsilon:
0.01
 Starting Laplace:
 Grid Size=  31
 Conductors set at V1=   100. V2=  -100.
 Relaxing with accuracy epsilon=   0.01
 1 err=   33.3333333
 2 err=   14.8148148
 3 err=   9.87654321
.......................
 110 err=   0.0106860904
 111 err=   0.0101182476
 112 err=   0.00958048937
```

In the example above, the program performs 112 sweeps until the error becomes `0.00958048937 < 0.01`. The results are stored in the file `data`. We can make a three dimensional plot of the function $V(i,j)$ with the `gnuplot` commands:

```
gnuplot> set pm3d
gnuplot> set hidden3d
gnuplot> set size ratio 1
gnuplot> splot "data" with lines
```

The results are shown in figure 7.7

## 7.7 Poisson Equation

This section contains a short discussion of the case where the space is filled with a continuous static charge distribution given by the charge density function $\rho(\vec{r})$. In this case the Laplace equation becomes the Poisson equation:

$$\nabla^2 V = \frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + \frac{\partial^2 V}{\partial z^2} = -4\pi\rho(x,y,z) \tag{7.13}$$

The equation on the lattice becomes

$$V(i,j) = \frac{1}{4}(V(i-1,j)+V(i+1,j)+V(i,j-1)+V(i,j+1)+\tilde{\rho}(i,j))\,, \tag{7.14}$$

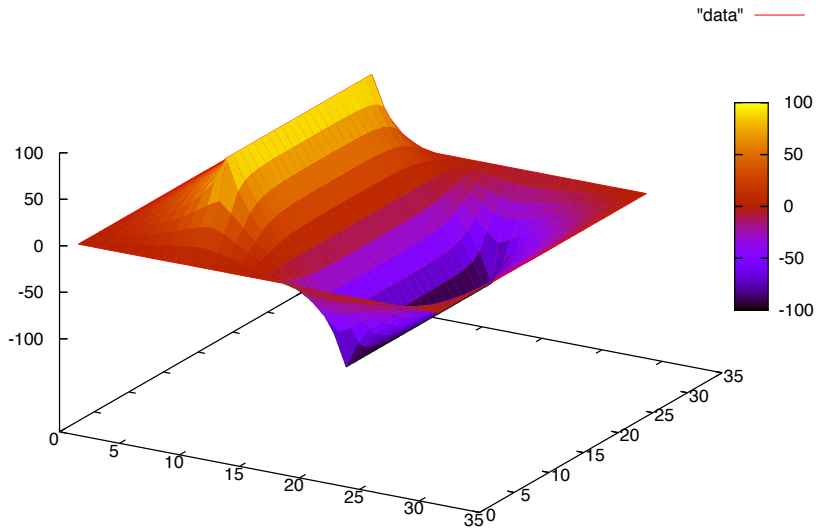



Figure 7.7: The solution of the equation (7.10) computed by the program `LaplaceEq.f90` for `L= 31, V1=100, V2=-100, epsilon=0.01`.

where[9] $\tilde{\rho}(i,j) = 4\pi a^2 \rho(i,j)$.

The program in the file `PoissonEq.f90` solves equation (7.14) for a uniform charge distribution (figure 7.10), where we have set $a = 1$. The reader is asked to reproduce this figure together with figures 7.8 and 7.9.

```
!*************************************************************
!set the boundary of a square to given potentials
!*************************************************************
program poisson_eq
 implicit none
 integer,parameter        :: P=51
 logical,dimension(P,P)   :: isConductor
 real(8),dimension(P,P)   :: V,rho
 real(8)                  :: V1,V2,V3,V4,Q,epsilon
 integer                  :: L

 L = P
 print *,'Enter V1,V2,V3,V4:'
 read  *, V1,V2,V3,V4
```

[9]Since $Q = \int \rho dA \approx \sum_{i,j} \rho a^2 = (1/4\pi) \sum_{i,j} \tilde{\rho}$. Therefore $\sum_{i,j} \tilde{\rho} \approx 4\pi Q$.

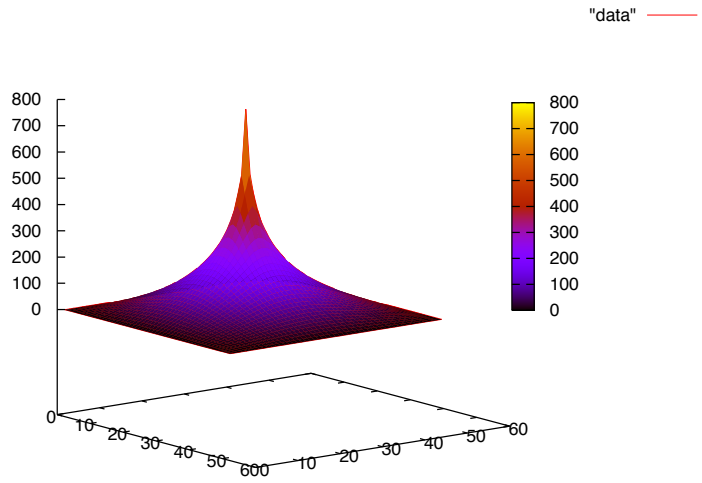



Figure 7.8: The solution of the equation (7.13) by the program in the file `Poisson.f90` for `L= 51`, `V= 0` on the boundary and the charge $4\pi Q = 1000$ all concentrated at one point.

```
 print *,'Enter 4*PI*Q:'
 read  *, Q
 print *,'Enter epsilon:'
 read  *, epsilon
 print *,'Starting Laplace:'
 print *,'Grid Size= ',L
 print *,'Boundaries set at V1= ',V1,' V2= ',V2,' V3= ',V3,&
      ' V4= ',V4,' and Q= ',Q
 print *,'Relaxing with accuracy epsilon= ',epsilon


 call initialize_lattice(V,isConductor,rho,L,V1,V2,V3,V4,Q)

 call laplace(V,isConductor,rho,L,epsilon)

 call print_results(V,L)

end program laplace_sq
!*********************************************************
subroutine &
initialize_lattice(V,isConductor,rho,L,V1,V2,V3,V4,Q)
!*********************************************************
```

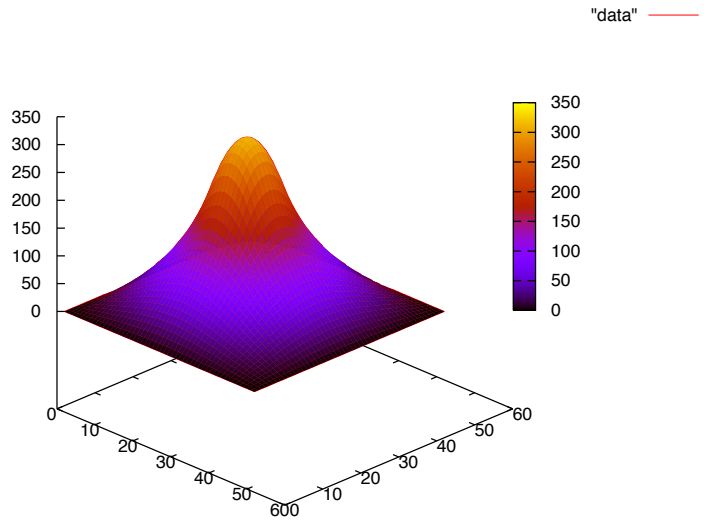



Figure 7.9: The solution of equation (7.13) by the program in the file `Poisson.f90` for `L= 51`, `V= 0` on the boundary and the charge $4\pi Q = 1000$ uniformly distributed in a small square with sides made of 10 lattice sites.

```
 implicit none
 integer                    :: L
 logical ,dimension(L,L)    :: isConductor
 real(8),dimension(L,L)     :: V,rho
 real(8)                    :: V1,V2,V3,V4,Q,Area
 integer                    :: i,j,L1,L2
!Initialize to 0 and .FALSE.
 V           =  0.0D0
 isConductor = .FALSE.
 rho         =  0.0D0
!We set the boundary to be a conductor:
 do i=1,L
  isConductor(1,i) = .TRUE.
  isConductor(i,1) = .TRUE.
  isConductor(L,i) = .TRUE.
  isConductor(i,L) = .TRUE.
  V          (1,i) =  V1
  V          (i,L) =  V2
  V          (L,i) =  V3
  V          (i,1) =  V4
 enddo
!We set the points with non-zero charge
```

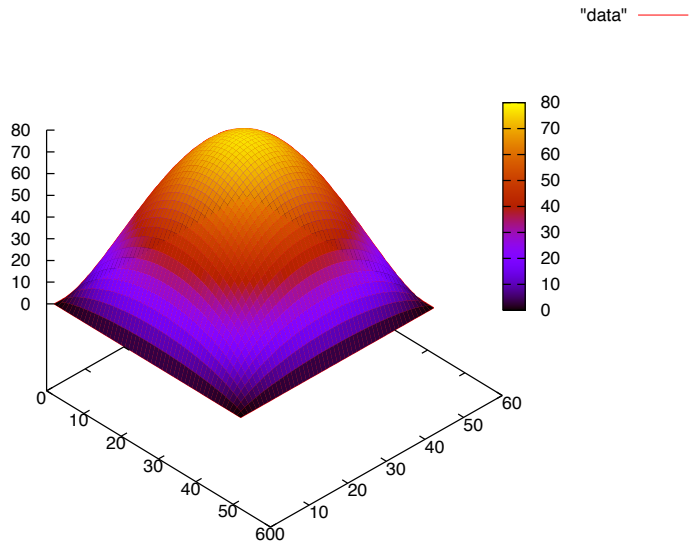



Figure 7.10: The solution of equation (7.13) by the program in the file `Poisson.f90` for `L= 51`, `V= 0` on the boundary and the charge $4\pi Q = 1000$ uniformly distributed on all internal lattice sites.

```
!A uniform distribution at a center square
 L1 = (L/2)-5
 L2 = (L/2)+5
 if(L1.LT.1) stop 'array rho out of bounds. Small L1'
 if(L2.GT.L) stop 'array rho out of bounds. Large L2'
 Area = (L2-L1+1)*(L2-L1+1)
 do j=L1,L2
  do i=L1,L2
   rho(i,j) = Q/Area !rho is \tilde\rho in notes
  enddo              !so Q is 4*PI*Q
 enddo

end subroutine initialize_lattice
!*************************************************************
subroutine laplace(V,isConductor,rho,L,epsilon)
!*************************************************************
 implicit none
 integer :: L
 logical,dimension(L,L) :: isConductor
 real(8),dimension(L,L) :: V,rho
 real(8)                :: epsilon
 integer                :: i,j,icount
```

```
 real(8)                  :: Vav,error,dV

 icount = 0
 do while (.TRUE.)
  error = 0.0D0
  do  j=2,L-1
   do i=2,L-1
!We change the voltage only for non conductors:
    if( .NOT. isConductor(i,j))then
     Vav = (V(i-1,j)+V(i+1,j)+V(i,j+1)+V(i,j-1)+rho(i,j))&
           *0.25D0
     dV  = DABS(V(i,j)-Vav)
     if(error .LT. dV ) error = dV !maximum error
     V(i,j) = Vav
    endif
   enddo
  enddo
  icount = icount + 1
  if( error .LT. epsilon) exit
 enddo
 print *,icount,' err= ',error

end subroutine laplace
!*************************************************************
subroutine  print_results(V,L)
!*************************************************************
 implicit none
 integer                  :: L
 real(8),dimension(L,L)  :: V
 integer                  :: i,j

 open(unit=11,file="data")
 do i=1,L
  do j =1,L
   write(11,*)i,j,V(i,j)
  enddo
  write (11,*)'' !empty line for gnuplot,separate isolines
 enddo

end subroutine print_results
```

In the bibliography the algorithm described above is called the Gauss–Seidel method. In this method, the right hand side of equation (7.14) uses the updated values of the potential in the calculation of $V(i,j)$ and $V(i,j)$ is immediately updated. In contrast, the Jacobi method uses the *old* values of the potential in the right hand side of (7.14) and the new value computed is stored in order to be used in the next sweep. The Gauss–Seidel method is superior to the Jacobi method as far as speed of

convergence is concerned. We can generalize Jacobi's method by defining the residual $R_{i,j}$ of equation (7.14)

$$R_{i,j} = V(i+1,j) + V(i-1,j) + V(i,j+1) + V(i,j-1) - 4V(i,j) + \tilde{\rho}(i,j)\,, \tag{7.15}$$

which vanishes when $V(i,j)$ is a solution of equation (7.14). Then, using $R_{i,j}$, Jacobi's method can be formulated as

$$V^{(n+1)}(i,j) = V^{(n)}(i,j) + \frac{1}{4}R^{(n)}_{i,j}\,, \tag{7.16}$$

where the quantities with index $(n)$ refer to the values of the potential during the $n$-th sweep. The successive overrelaxation (SOR) method is given by:

$$V^{(n+1)}(i,j) = V^{(n)}(i,j) + \frac{\omega}{4}R^{(n)}_{i,j}\,. \tag{7.17}$$

When $\omega < 1$ we have "underrelaxation" and we obtain slower convergence than the Jacobi method. When $1 < \omega < 2$ we have "overrelaxation" and an appropriate choice of $\omega$ can lead to an improvement compared to the Jacobi method. When $\omega > 2$ SOR diverges. Further study of the SOR methods is left as an exercise to the reader.

## 7.8 Problems

7.1 Reproduce the figures with the electric field lines and equipotential lines shown in section 7.2.

7.2 Take the charge distributions that you used in the previous problems, make all the charges to be positive and remake the figures of the field lines and the equipotential lines. Then repeat by taking half of the charges to be twice in magnitude than the others.

7.3 The program `ELines.f90` gets stuck when you apply it on a charge distribution of four equal charges located at the vertices of a square. How can you correct this pathology?

7.4 Make the necessary changes to the program in the file `ELines.f90` so that the number of field lines starting near a charge $q$ is proportional to $q$.

7.5 Improve the program in `EPotential.f90` so that the equipotential lines are drawn with a density proportional to the magnitude of the electric field.
Hint:

(a) Write a subroutine that calculates the potential $V(x, y)$ at the point $(x, y)$.

(b) From each point charge draw a line in the radial direction and calculate the potential on points that are at small distance $\Delta l$ from each other.

(c) Calculate the maximum/minimum value of the potential $V_{max}/V_{min}$ and use them in order to choose the values of the potential on the equipotential lines that you plan to draw. If e.g. you choose to draw 5 equipotential lines, take $\delta V = (V_{max} - V_{min})/4$ and $V_i = V_{min} + i\delta V \;\; i = 0, \ldots, 4$.

(d) Repeat the second step. When the potential at a point takes approximately one of the values $V_i$ chosen in the previous step, draw an equipotential line from that point.

7.6 Compute the electric potential using the program in the file `LaplaceEq.f90` for

(a) `L= 31, V1=100, V2=100`

(b) `L= 31, V1=100, V2=0`

and construct the corresponding plot for $V(i,j)$.

7.7 Compute the electric potential using the program in the file `LaplaceEq.f90` for

(a) `V1=100, V2=100`

(b) `V1=100, V2=100`

(c) `V1=100, V2=0`

for `L=31,61,121,241,501` and construct the corresponding plot for $V(i,j)$. Vary `epsilon=0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001`. What is the dependence of the number of sweeps $N$ on `epsilon`? Make the plot of $N$(`epsilon`). Put the points and curves of $N$(`epsilon`) for all values of `L` on the same plot.

7.8 Compute the electrostatic potential of a square conductor when the potential on each side is `V1, V2, V3, V4`. Repeat what you did in the previous problem for

(a) `V1=10, V2=5, V3=10, V4= 5`

(b) `V1=10, V2=0, V3=0, V4= -10`

(c) `V1=10, V2=0, V3=0, V4= 0`

7.9 Compute the electrostatic potential of a system of square conductors where the one is inside the other as shown in figure 7.11. The side of each conductor has `L1, L2` sites respectively and the value of the potential is `V1,V2` respectively. Take `L2= L1/5` and repeat the steps in the previous problem for `V1=10, V2=-10` and `L1= 25, 50, 100, 200`.

7.10 Perform a numerical computation of the capacitance $C = Q/V$ of the system of conductors of the previous problem when $V_1 = V$, $V_2 = -V$. In order to calculate the charge $Q$, compute the surface charge density $\sigma$ using the equation

$$\sigma = \frac{E_n}{4\pi} ,$$

where $E_n$ is the perpendicular component of the electric field on the surface. Use the approximation

$$E_n = -\frac{\delta V}{\delta r} ,$$

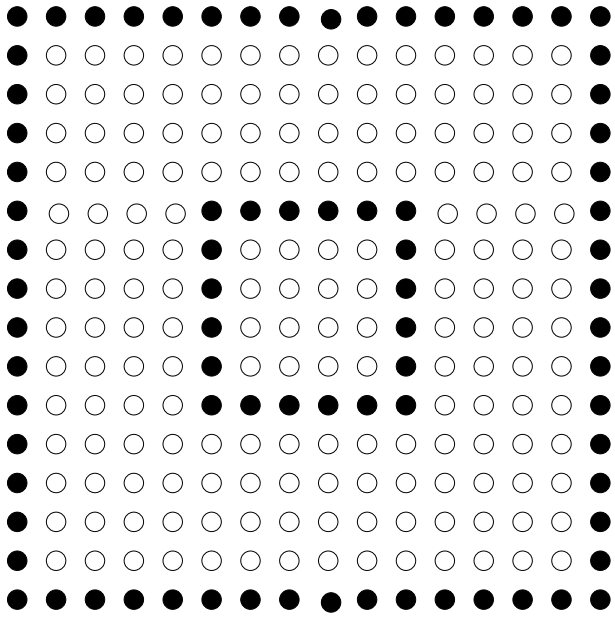


Figure 7.11: The square conductors described in problem 7.9.

where $\delta V$ is the potential difference between a point on the conductor and its nearest neighbor. By integrating (i.e. summing) you can estimate the total charge on each conductor. If these are opposite and their absolute value is $Q$, then the capacitance can be calculated from the equation $C = Q/V$. Perform the calculation described above for $V = 10$ and `L1=25, 75`.

7.11 In the system of the previous problem compute the function $Q(V)$. Verify that the capacitance is independent of $V$. Use `L1=25,50`, `V1= -V2 =1, 2, 5, 10, 15, 20, 25`.

7.12 Reproduce figures 7.8, 7.9 and 7.10. Compare the result of the first case with the known solution of a point charge in empty space.

7.13 Introduce the lattice spacing $a$ in the corresponding equations in the program in the file `PoissonEq.f90`. Set the length of each side to be $l = 1$ and print the results in the file `data` as $(x_i, y_i, V(x_i, y_i))$ instead of $(i, j, V(i, j))$. Take `L=51,101,151,201,251` and plot $V(x, y)$ in the square $0 < x < 1$, $0 < y < 1$. Study the convergence of the solutions by plotting the section $V(x, 1/2)$ for each `L`.

7.14 Write a program that implements the SOR algorithm given by equa-

tion (7.16) for the problem solved in `LaplaceEq.f90`. Compare the speed of convergence of SOR with that of the Gauss-Seidel method for $L = 51$, $\omega = 1.0, 0.9, 0.8, 0.6, 0.4, 0.2$. What happens when $\omega > 1$?

7.15 Write a program that implements the SOR algorithm given by equation (7.16) for the problem solved in `PoissonEq.f90`. Compare the speed of convergence of SOR with that of the Gauss-Seidel method for $L = 51$, $\omega = 1.0, 0.9, 0.8, 0.6, 0.4, 0.2$. What happens when $\omega > 1$?

# Chapter 8

# Diffusion Equation

## 8.1 Introduction

The diffusion equation is related to the study of random walks. Consider a particle moving on a line (one dimension) performing a random walk. The motion is stochastic and the kernel

$$K(x, x_0; t)\,, \tag{8.1}$$

is interpreted as the probability density to observe the particle at position $x$ at time $t$ if the particle is at $x_0$ at $t = 0$. The equation that determines $K(x, x_0; t)$ is

$$\frac{\partial K(x, x_0; t)}{\partial t} = D\,\frac{\partial^2 K(x, x_0; t)}{\partial x^2}\,, \tag{8.2}$$

which is the diffusion equation. The coefficient $D$ depends on the details of the system that is studied. For example, for the Brownian motion of a dust particle in a fluid which moves under the influence of random collisions with the fluid particles, we have that $D = kT/\gamma$, where $T$ is the (absolute) temperature of the fluid, $\gamma$ is the friction coefficient[1] of the particle in the fluid and $k$ is the Boltzmann constant.

Usually the initial conditions are chosen so that at $t = 0$ the particle is localized at one point $x_0$, i.e.[2]

$$K(x, x_0; 0) = \delta(x - x_0)\,. \tag{8.3}$$

[1]For a spherical particle of radius $R$ in a Newtonian liquid with viscosity $\eta$ we have that $\gamma = 6\pi\eta R$.

[2] $\delta(x - x_0)$ is the Dirac delta "function". It can be defined from the requirement that for every function $f(x)$ we have that $\int_{-\infty}^{+\infty} f(x)\delta(x - x_0)\,dx = f(x_0)$. Obviously we also have that $\int_{-\infty}^{+\infty} \delta(x - x_0)\,dx = 1$. Intuitively one can think of it as a function that is almost zero everywhere except in an infinitesimal neighborhood of $x_0$.

The interpretation of $K(x, x_0; t)$ as a probability density implies that for *every* $t$ we should have that[3]

$$\int_{-\infty}^{+\infty} K(x, x_0; t)\, dx = 1\,. \tag{8.4}$$

It is not obvious that this relation can be imposed for every instant of time. Even if $K(x, x_0; t)$ is normalized so that (8.4) holds for $t = 0$, the time evolution of $K(x, x_0; t)$ is governed by equation (8.2) which can spoil equation (8.4) at later times.

If we impose equation (8.4) at $t = 0$, then it will hold at all times if

$$\frac{d}{dt}\int_{-\infty}^{+\infty} K(x, x_0; t) dx = 0\,. \tag{8.5}$$

By taking into account that $\frac{d}{dt}\int_{-\infty}^{+\infty} K(x, x_0; t)dx = \int_{-\infty}^{+\infty} \frac{\partial K(x,x_0;t)}{\partial t} dx$ and that $\frac{\partial K(x,x_0;t)}{\partial t} = D\frac{\partial^2 K(x,x_0;t)}{\partial x^2}$ we obtain

$$\begin{aligned}\frac{d}{dt}\int_{-\infty}^{+\infty} K(x, x_0; t)dx &= D\int_{-\infty}^{+\infty} \frac{\partial}{\partial x}\left(\frac{\partial K(x, x_0; t)}{\partial x}\right) dx \\ &= D\left.\frac{\partial K(x, x_0; t)}{\partial x}\right|_{x\to+\infty} - D\left.\frac{\partial K(x, x_0; t)}{\partial x}\right|_{x\to-\infty}\,.\end{aligned} \tag{8.6}$$

The above equation tells us that for functions for which the right hand side vanishes, the normalization condition will be valid for all $t > 0$.

A careful analysis of equation (8.2) gives that the asymptotic behavior of $K(x, x_0; t)$ for small times is

$$K(x, x_0; t) \sim \frac{\mathrm{e}^{-\frac{|x-x_0|^2}{4Dt}}}{t^{d/2}} \sum_{i=0}^{\infty} a_i(x, x_0) t^i\,. \tag{8.7}$$

This relation shows that diffusion is isotropic (the same in all directions) and that the probability of detecting the particle drops exponentially with the distance squared from the initial position of the particle. This relation cannot hold for all times, since for large enough times the probability of detecting the particle will be the same everywhere[4].

[3]Alternatively, if $K(x, x_0; t)$ is interpreted as e.g. the mass density of a drop of ink of mass $m_{ink}$ inside a transparent liquid, we will have that $\int_{-\infty}^{+\infty} K(x, x_0; t)\, dx = m_{ink}$ και $K(x, x_0; 0) = m_{ink}\delta(x - x_0)$.

[4]Remember the analogy of an ink drop diffusing in a transparent liquid. After long enough time, the ink is homogeneously dissolved in the liquid.

The return probability of the particle to its initial position is

$$P_R(t) = K(x_0, x_0; t) \sim \frac{1}{t^{d/2}} \sum_{i=0}^{\infty} a_i(x_0, x_0) t^i \,. \tag{8.8}$$

The above relation defines the *spectral dimension* $d$ of space. $d = 1$ in our case.

The expectation value of the distance squared of the particle at time $t$ is easily calculated[5]

$$\langle r^2 \rangle = \langle (x - x_0)^2 \rangle(t) = \int_{-\infty}^{+\infty} (x - x_0)^2 K(x, x_0; t)\, dx \sim 2Dt \,. \tag{8.9}$$

This equation is very important. It tells us that the random walk (Brownian motion) is not a classical motion but it can only be given a stochastic description: A classical particle moving with constant velocity $v$ so that $x - x_0 \sim vt$ results in $r^2 \sim t^2$.

In the following sections we take[6] $D = 1$ and define

$$u(x, t) \equiv K(x - x_0, x_0; t) \,. \tag{8.10}$$

## 8.2 Heat Conduction in a Thin Rod

Consider a thin rod of length $L$ and let $T(x, t)$ be the temperature distribution within the rod at time $t$. The two ends of the rod are kept at constant temperature $T(0, t) = T(L, t) = T_0$. If the initial temperature distribution in the rod is $T(x, 0)$, then the temperature distribution at all times is determined by the diffusion equation

$$\frac{\partial T(x, t)}{\partial t} = \alpha \frac{\partial^2 T(x, t)}{\partial x^2} \,, \tag{8.11}$$

where $\alpha = k/(c_p \rho)$ is the thermal diffusivity, $k$ is the thermal conductivity, $\rho$ is the density and $c_p$ is the specific heat of the rod.

Define

$$u(x, t) = \frac{T(xL, \frac{L^2}{\alpha} t) - T_0}{T_0} \,, \tag{8.12}$$

where $x \in [0, 1]$. The function $u(x, t)$, giving the fraction of the temperature difference to the temperature at the ends of the rod, is dimensionless and

$$u(0, t) = u(1, t) = 0 \,. \tag{8.13}$$

[5] $\int_0^\infty dr\, r^n \mathrm{e}^{-r^2/4Dt} = 2^n \Gamma(\frac{n+1}{2})(Dt)^{\frac{n+1}{2}}$.

[6] According to equation (8.2) this amounts to taking $t \to Dt$.

These are called Dirichlet boundary conditions[7].

Equation (8.11) becomes

$$\frac{\partial u(x,t)}{\partial t} = \frac{\partial^2 u(x,t)}{\partial x^2} \tag{8.14}$$

Equation (8.6) becomes

$$\frac{d}{dt}\int_0^1 u(x,t)dx = \left.\frac{\partial u}{\partial x}\right|_{x=1} - \left.\frac{\partial u}{\partial x}\right|_{x=0} \tag{8.15}$$

The relation above cannot be equal to zero at all times due to the boundary conditions (8.13). This can be easily understood with an example. Suppose that

$$u(x,0) = \sin(\pi x)\,, \tag{8.16}$$

then it is easy to confirm that the boundary conditions are satisfied and that the function

$$u(x,t) = \sin(\pi x)\mathrm{e}^{-\pi^2 t}\,, \tag{8.17}$$

is the solution to the diffusion equation. It is easy to see that

$$\int_0^1 u(x,t)dx = \frac{2}{\pi}\mathrm{e}^{-\pi^2 t}$$

drops exponentially with time and that

$$\frac{d}{dt}\int_0^1 u(x,t)dx = -2\pi\mathrm{e}^{-\pi^2 t}\,,$$

which is in agreement with equations (8.15).

The exponential drop of the magnitude of $u(x,t)$ is in agreement with the expectation that the rod will have constant temperature at long times, which will be equal to the temperature at its ends ($\lim_{t\to+\infty} u(x,t) = 0$).

## 8.3 Discretization

The numerical solution of equation (8.14) will be computed in the interval $x \in [0,1]$ for $t \in [0,t_f]$. The problem will be defined on a two dimensional discrete lattice and the differential equation will be approximated by finite difference equations.

[7] If the derivative $\partial u/\partial x$ was given as a boundary condition instead, then we would have Neumann boundary conditions.

The lattice is defined by $N_x$ spatial points $x_i \in [0,1]$

$$x_i = 0 + (i-1)\Delta x \quad i = 1, \ldots, N_x \,, \tag{8.18}$$

where the $N_x - 1$ intervals have the same width

$$\Delta x = \frac{1-0}{N_x - 1} \,, \tag{8.19}$$

and by the $N_t$ time points $t_j \in [0, t_f]$

$$t_j = 0 + (j-1)\Delta t \quad j = 1, \ldots, N_t \,, \tag{8.20}$$

where the $N_t - 1$ time intervals have the same duration

$$\Delta t = \frac{t_f - 0}{N_t - 1} \,. \tag{8.21}$$

We note that the ends of the intervals correspond to

$$x_1 = 0 \,, \quad x_{N_x} = 1 \,, \quad t_1 = 0 \,, \quad t_{N_t} = t_f \,. \tag{8.22}$$

The function $u(x,t)$ is approximated by its values on the $N_x \times N_t$ lattice

$$u_{i,j} \equiv u(x_i, t_j) \,. \tag{8.23}$$

The derivatives are replaced by the finite differences

$$\frac{\partial u(x,t)}{\partial t} \approx \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j)}{\Delta t} \equiv \frac{1}{\Delta t} \left( u_{i,j+1} - u_{i,j} \right) \,, \tag{8.24}$$

$$\begin{aligned} \frac{\partial^2 u(x,t)}{\partial x^2} &\approx \frac{u(x_i + \Delta x, t_j) - 2u(x_i, t_j) + u(x_i - \Delta x, t_j)}{(\Delta x)^2} \\ &\equiv \frac{1}{(\Delta x)^2} \left( u_{i+1,j} - 2u_{i,j} + u_{i-1,j} \right) \,. \end{aligned} \tag{8.25}$$

By equating both sides of the above relations according to (8.14), we obtain the dynamic evolution of $u_{i,j}$ in time

$$u_{i,j+1} = u_{i,j} + \frac{\Delta t}{(\Delta x)^2} \left( u_{i+1,j} - 2u_{i,j} + u_{i-1,j} \right) \,. \tag{8.26}$$

This is a one step iterative relation in time. This is very convenient, because one does not need to store the values $u_{i,j}$ for all $j$ in the computer memory.

The second term (the "second derivative") in (8.26) contains only the nearest neighbors $u_{i\pm1,j}$ of the lattice point $u_{i,j}$ at a given time slice $t_j$. Therefore it can be used for all $i = 2, \ldots, N_x - 1$. The relations (8.26) are not needed for the points $i = 1$ and $i = N_x$ since the values $u_{1,j} = u_{N_x,j} = 0$ are kept constant.

The parameter

$$\frac{\Delta t}{(\Delta x)^2} \tag{8.27}$$

determines the time evolution in the algorithm. It is called the Courant parameter and in order to have a time evolution without instabilities it is necessary to have

$$\frac{\Delta t}{(\Delta x)^2} < \frac{1}{2}\,. \tag{8.28}$$

This condition will be checked in our analysis empirically.

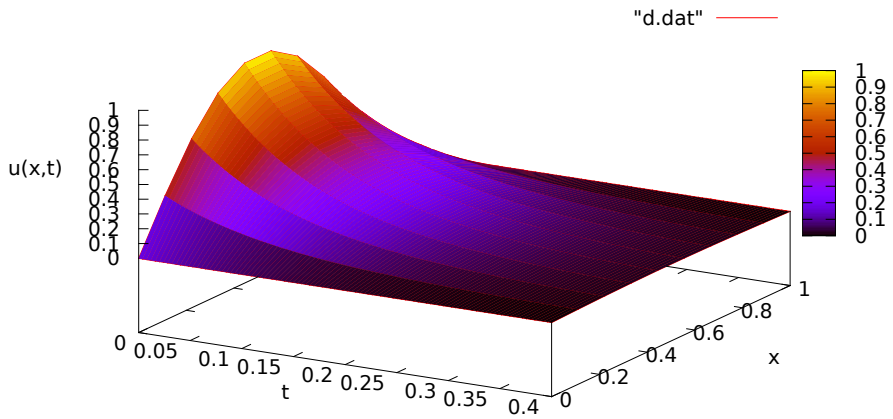



Figure 8.1: The function $u(x,t)$ for `Nx=10, Nt=100, tf= 0.4`.

## 8.4 The Program

The fact that equation (8.26) is a one time step iterative relation, leads to a substantial simplification of the structure of the program. Because of this, at each time step, it is sufficient to store the values of the second term (the "second derivative") in one array. This array will be used in order

to update the values of $u_{i,j}$. Therefore we will define only two arrays $u_i$, $i = 1, \ldots, N_x$ and $(\partial^2 u/\partial x^2)_i$, $i = 1, \ldots, N_x$ which store the values of $u_{i,j}$ and $\Delta t/(\Delta x)^2(u_{i+1,j} - 2u_{i,j} + u_{i-1,j})$ at time $t_j$ respectively. In the program listed below, the names of these arrays are `u(P)` and `d2udx2(P)`.

The data is stored in the array positions `u(1) ... u(Nx)` and `d2udx2(1) ... d2udx2(Nx)` and the parameter `P` is taken large enough so that `Nx` is always smaller than `P`.

The user enters the $N_x =$ `Nx`, $N_t =$`Nt` and $t_f =$`tf` interactively. The values of $\Delta x$, $\Delta t$ and $\Delta t/\Delta x^2 =$ `courant` are calculated during the initialization.

On exit, we obtain the results in the file `d.dat` which contains $(t_j, x_i, u_{i,j})$ in three columns. When a time slice is printed, the program prints an empty line so that the output is easily read by the three dimensional plotting function `splot` of `gnuplot`.

The program is in the file `diffusion.f90` and is listed below:

```
!=========================================================
! 1-dimensional Diffusion Equation with simple
! Dirichlet boundary conditions u(0,t)=u(1,t)=0
! 0<= x <= 1 and 0<= t <= tf
!
! We set initial condition u(x,t=0) that satisfies
! the given boundary conditions.
! Nx is the number of points in spatial lattice:
! x = 0 + (j-1)*dx, j=1,...,Nx and dx = (1-0)/(Nx-1)
! Nt is the number of points in temporal lattice:
! t = 0 + (j-1)*dt, j=1,...,Nt and dt = (tf-0)/(Nt-1)
!
! u(x,0) = sin(pi*x) tested against analytical solution
! u(x,t) = sin(pi*x)*exp(-pi*pi*t)
!
!=========================================================
program diffusion_1d
 implicit none
 integer,parameter     :: P =100000  ! Max no of points
 real(8),parameter     :: PI=3.1415926535897932D0
 real(8),dimension(P) :: u, d2udx2
 real(8) :: t,x,dx,dt,tf,courant
 integer Nx,Nt,i,j
! --- Input:
 print *, '# Enter: Nx, Nt, tf: (P= ',P,' Nx must be < P)'
 read  *, Nx,Nt,tf
 if(Nx .ge. P) stop 'Nx >= P'
 if(Nx .le. 3) stop 'Nx <= 3'
 if(Nt .le. 2) stop 'Nt <= 2'
```

```
! --- Initialize:
 dx      = 1.0D0/(Nx-1)
 dt      = tf    /(Nt-1)
 courant = dt/dx**2
 print * ,'# 1d Diffusion Equation: 0<=x<=1, 0<=t<=tf'
 print * ,'# dx= ',dx,' dt= ',dt,' tf= ', tf
 print * ,'# Nx= ',Nx,' Nt= ',Nt
 print * ,'# Courant Number= ',courant
 if(courant .gt. 0.5D0) print *,'# WARNING: courant > 0.5'
 open(unit=11,file='d.dat') ! data file
! --- Initial condition at t=0 ------------------------------
!u(x,0) = sin( pi x)
 do i= 1, Nx
  x    = (i-1)*dx
  u(i) = sin(PI*x)
 enddo
 u(1)  = 0.0d0
 u(Nx) = 0.0d0
 do i= 1,Nx
  x    = (i-1)*dx
  write(11,*) 0.0D0, x, u(i)
 enddo
 write(11,*)' '
! ------------------------------------------------------------
! --- Calculate time evolution:
 do j=2,Nt
  t = (j-1)*dt
! ----- second derivative:
  do i=2,Nx-1
   d2udx2(i) = courant*(u(i+1)-2.0D0*u(i)+u(i-1))
  enddo
! ----- update:
  do i=2,Nx-1
   u(i) = u(i) + d2udx2(i)
  enddo
  do i=1,Nx
   x =  (i-1)*dx
   write(11,*) t, x, u(i)
  enddo
  write(11,*)' '

 enddo ! do j=2,Nt

 close(11)
end program diffusion_1d
```

## 8.5 Results

The compilation and running of the program can be done with the commands:

```
> gfortran diffusion.f90 -o d
> echo "10 100 0.4" | ./d
 # Enter: Nx, Nt, tf: (P= 100000  Nx must be < P)
 # 1d Diffusion Equation: 0<=x<=1, 0<=t<=tf
 # dx= 0.11111111111111110 dt= 4.04040404040404040E-3 tf= 0.4
 # Nx= 10                  Nt= 100
 # Courant Number=   0.32727272727272733
```

The input to the program `./d` is read from the `stdin` and it is given by the `stdout` of the command `echo` through a pipe, as shown in the second line in the listing above. The lines that follow are the standard output `stdout` of the program.

The three dimensional plot of the function $u(x,t)$ can be made with the `gnuplot` commands:

```
gnuplot> set pm3d
gnuplot> set hidden3d
gnuplot> splot "d.dat" with lines
gnuplot> unset pm3d
```

In order to make the plot of $u(x,t)$ for a fixed value of $t$ we first note that an empty line in the file `d.dat` marks a change in time. The following `awk` program counts the empty lines of `d.dat` and prints only the lines when the number of empty lines that have been encountered so far is equal to 3. The counter `n=0, 1, ..., Nt-1` determines the value of $t_j = t_{n-1}$. We save the results in the file `tj` which can be plotted with `gnuplot`. We repeat as many times as we wish:

```
> awk 'NF<3{n++}n==3  {print}' d.dat > tj
gnuplot> plot "tj" using 2:3 with lines
```

The above task can be completed without creating the intermediate file `tj` by using the `awk` filter within `gnuplot`. For example, the commands

```
gnuplot> ! echo "10 800 2" | ./d
gnuplot> plot   "<awk 'NF<3{n++}n==3  {print}' d.dat" u 2:3 w l
gnuplot> replot "<awk 'NF<3{n++}n==6  {print}' d.dat" u 2:3 w l
gnuplot> replot "<awk 'NF<3{n++}n==10 {print}' d.dat" u 2:3 w l
```

```
gnuplot> replot "<awk 'NF<3{n++}n==20 {print}' d.dat" u 2:3 w l
gnuplot> replot "<awk 'NF<3{n++}n==30 {print}' d.dat" u 2:3 w l
gnuplot> replot "<awk 'NF<3{n++}n==50 {print}' d.dat" u 2:3 w l
gnuplot> replot "<awk 'NF<3{n++}n==100{print}' d.dat" u 2:3 w l
```

run the program for `Nx=10, Nt=800, tf= 2` and construct the plot in figure 8.2

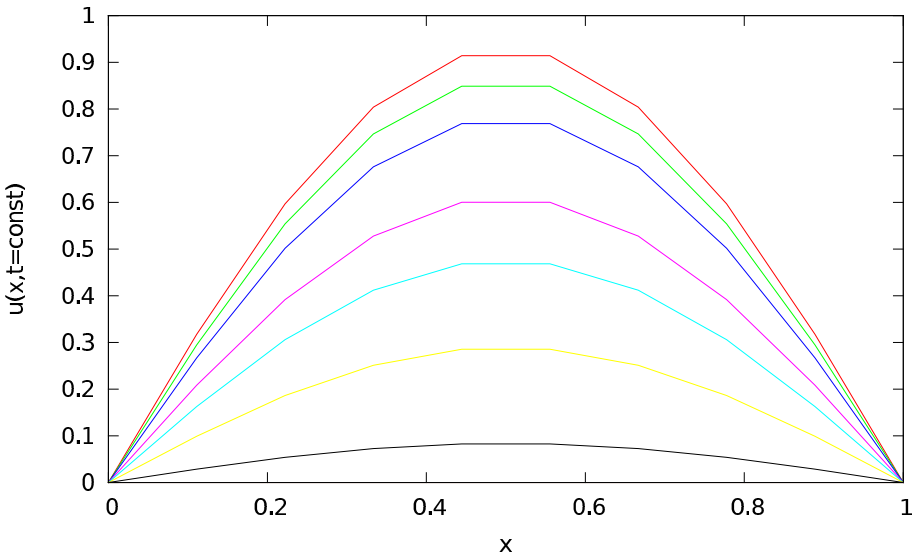



Figure 8.2: The function $u(x,t)$ for `Nx=10, Nt=800, tf= 2` for different values of the time $t_j$. We take $j = 4, 7, 11, 21, 31, 51, 101$ and observe that the function $u(x,t)$ decreases then $j$ increases.

It is instructive to compare the results with the known solution $u(x,t) = \sin(\pi x)\mathrm{e}^{-\pi^2 t}$. We compute the relative error

$$\frac{u_{i,j} - u(x_i, t_j)}{u_{i,j}},$$

which can be done within `gnuplot` with the commands:

```
gnuplot> du(x,y,z) = (z - sin(pi*x)*exp(-pi*pi*y))/z
gnuplot> plot "<awk 'NF<3{n++}n==2  ' d.dat" u 2:(du($2,$1,$3))
gnuplot> plot "<awk 'NF<3{n++}n==6  ' d.dat" u 2:(du($2,$1,$3))
gnuplot> plot "<awk 'NF<3{n++}n==20 ' d.dat" u 2:(du($2,$1,$3))
gnuplot> plot "<awk 'NF<3{n++}n==200' d.dat" u 2:(du($2,$1,$3))
gnuplot> plot "<awk 'NF<3{n++}n==600' d.dat" u 2:(du($2,$1,$3))
gnuplot> plot "<awk 'NF<3{n++}n==780' d.dat" u 2:(du($2,$1,$3))
```

The results can be seen in figure 8.3.

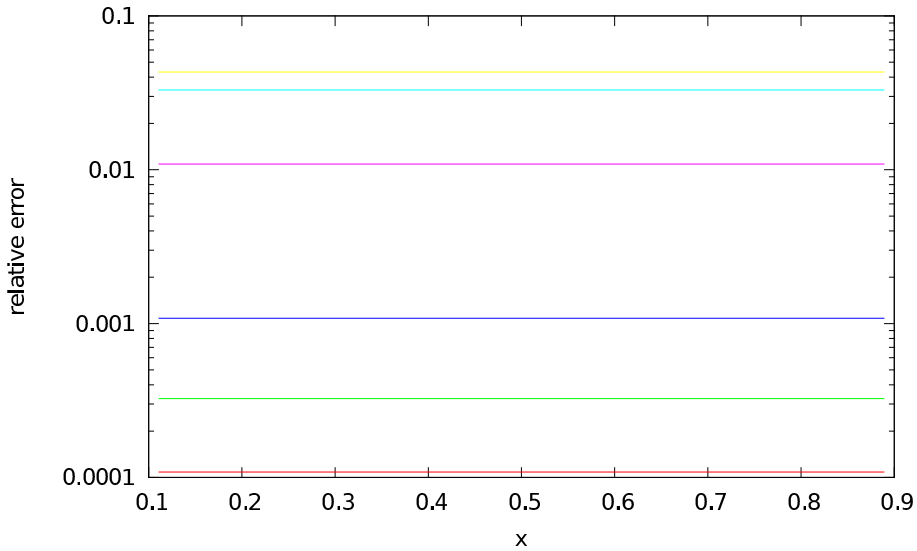



Figure 8.3: The absolute value of the relative error of the numerical computation for `Nx=10, Nt=800, tf= 2` for different times $t_j$. We take $j = 3, 7, 21, 201, 601, 781$ and observe that the relative error increases with $j$.

## 8.6 Diffusion on the Circle

In order to study the kernel $K(x, x_0; t)$ for the diffusion, or random walk, problem, we should impose the normalization condition (8.4) for all times. In the case of the function $u(x, t)$ defined for $x \in [0, 1]$ the relation becomes

$$\int_0^1 u(x, t)\, dx = 1\,. \tag{8.29}$$

In order to maintain this relation at all times, it is necessary that the right hand side of equation (8.15) is equal to 0. One way to impose this condition is to study the diffusion problem on the circle. If we parametrize the circle using the variable $x \in [0, 1]$, then the points $x = 0$ and $x = 1$ are identified and we obtain

$$u(0, t) = u(1, t)\,, \qquad \frac{\partial u(0, t)}{\partial x} = \frac{\partial u(1, t)}{\partial x}\,. \tag{8.30}$$

The second relation in the above equations makes the right hand side of equation (8.15) to vanish. Therefore if $\int_0^1 u(x, 0)\, dx = 1$, we obtain $\int_0^1 u(x, t)\, dx = 1$, $\forall t > 0$.

Using the above assumptions, the discretization of the differential equation is done exactly as in the problem of heat conduction. Instead

of keeping the values $u(0,t) = u(1,t) = 0$, we apply equation (8.26) also for the points $x_1$, $x_{N_x}$. In order to take into account the cyclic topology we take

$$u_{1,j+1} = u_{1,j} + \frac{\Delta t}{(\Delta x)^2}\left(u_{2,j} - 2u_{1,j} + u_{N_x,j}\right) , \tag{8.31}$$

and

$$u_{N_x,j+1} = u_{i,j} + \frac{\Delta t}{(\Delta x)^2}\left(u_{1,j} - 2u_{N_x,j} + u_{N_x-1,j}\right) , \tag{8.32}$$

since the neighbor to the right of the point $x_{N_x}$ is the point $x_1$ and the neighbor to the left of the point $x_1$ is the point $x_{N_x}$. For the rest of the points $i = 2, \ldots, N_x - 1$ equation (8.26) is applied normally.

The program that implements the problem described above can be found in the file `diffusionS1.f90`. The boundary conditions (8.30) are enforced in the lines

```
nnr = i+1
if(nnr .gt. Nx) nnr = 1
nnl = i-1
if(nnl .lt. 1 ) nnl = Nx
d2udx2(i) = courant*(u(nnr)-2.0D0*u(i)+u(nnl))
```

The initial conditions at $t = 0$ are chosen so that the particle is located at $x_{N_x/2}$. For each instant of time we perform measurements in order to verify the equations (8.4) and (8.9) and the fact that $\lim_{t\to+\infty} u(x,t) =$ const.

The variable `prob` $= \sum_{i=1}^{N_x} u_{i,j}$ and we should check that its value is conserved and is always equal to 1.

The variable `r2` $= \sum_{i=1}^{N_x}(x_i - x_{N_x/2})^2 u_{i,j}$ is a discrete estimator of the expectation value of the distance squared from the initial position. For small enough times it should follow the law given by equation (8.9).

These variables are written to the file `e.dat` together with the values $u_{N_x/2,j}$, $u_{N_x/4,j}$ και $u_{1,j}$. The latter are measured in order to check if for large enough times they obtain the *same* constant value according to the expectation $\lim_{t\to+\infty} u(x,t) =$ const.

The full code is listed below:

```
!=========================================================
! 1-dimensional Diffusion Equation with
! periodic boundary conditions u(0,t)=u(1,t)
! 0<= x <= 1 and 0<= t <= tf
!
! We set initial condition u(x,t=0) that satisfies
```

```
! the given boundary conditions.
! Nx is the number of points in spatial lattice:
! x = 0 + (j-1)*dx, j=1,...,Nx and dx = (1-0)/(Nx-1)
! Nt is the number of points in temporal lattice:
! t = 0 + (j-1)*dt, j=1,...,Nt and dt = (tf-0)/(Nt-1)
!
! u(x,0) = \delta_{x,0.5}
!
!=========================================================
program diffusion_1d
 implicit none
 integer,parameter     :: P =100000  ! Max no of points
 real(8),parameter     :: PI=3.1415926535897932D0
 real(8),dimension(P) :: u, d2udx2
 real(8) :: t,x,dx,dt,tf,courant,prob,r2,x0
 integer Nx,Nt,i,j,nnl,nnr
! --- Input:
 print *, '# Enter: Nx, Nt, tf: (P= ',P,' Nx must be < P)'
 read  *, Nx,Nt,tf
 if(Nx .ge. P) stop 'Nx >= P'
 if(Nx .le. 3) stop 'Nx <= 3'
 if(Nt .le. 2) stop 'Nt <= 2'
! --- Initialize:
 dx      = 1.0D0/(Nx-1)
 dt      = tf   /(Nt-1)
 courant = dt/dx**2
 print * ,'# 1d Diffusion Equation on S1: 0<=x<=1, 0<=t<=tf'
 print * ,'# dx= ',dx,' dt= ',dt,' tf= ', tf
 print * ,'# Nx= ',Nx,' Nt= ',Nt
 print * ,'# Courant Number= ',courant
 if(courant .gt. 0.5D0) print *,'# WARNING: courant > 0.5'
 open(unit=11,file='d.dat') ! data file
 open(unit=12,file='e.dat') ! data file
! --- Initial condition at t=0 ------------------------------
 do i= 1, Nx
  x      = (i-1)*dx
  u(i)   = 0.0D0
 enddo
 u(Nx/2) = 1.0D0
 do i= 1,Nx
  x      = (i-1)*dx
  write(11,*) 0.0D0, x, u(i)
 enddo
 write(11,*)' '
! ---------------------------------------------------------
! --- Calculate time evolution:
 do j=2,Nt
  t = (j-1)*dt
! ----- second derivative:
```

```
  do i=1,Nx
   nnr = i+1
   if(nnr .gt. Nx) nnr = 1
   nnl = i-1
   if(nnl .lt. 1 ) nnl = Nx
   d2udx2(i) = courant*(u(nnr)-2.0D0*u(i)+u(nnl))
  enddo
! ------ update:
  prob = 0.0D0
  r2   = 0.0D0
  x0   = ((Nx/2)-1)*dx !original position
  do i=1,Nx
   x    =  (i-1)*dx
   u(i) = u(i) + d2udx2(i)
   prob = prob + u(i)
   r2   = r2   + u(i)*(x-x0)*(x-x0)
  enddo
  do i=1,Nx
   x =  (i-1)*dx
   write(11,*) t, x, u(i)
  enddo
  write(11,*)' '
  write(12,*) 'pu ',t, prob,r2,u(Nx/2),u(Nx/4),u(1)
 enddo ! do j=2,Nt

 close(11)
end program diffusion_1d
```

## 8.7 Analysis

For each moment of time, the program writes the following quantities to the file `e.dat`:

$$U_j = \sum_{i=1}^{N_x} u_{i,j} \tag{8.33}$$

which is an estimator of (8.29) and we expect to obtain $U_j = 1$ for all $j$,

$$\langle r^2 \rangle_j = \sum_{i=1}^{N_x} u_{i,j} (x_i - x_{N_x/2})^2 \tag{8.34}$$

which is an estimator of (8.9) for which we expect to obtain

$$\langle r^2 \rangle_j \sim 2t_j \,, \tag{8.35}$$

for small times as well as the values of $u_{N_x/2,j}$, $u_{N_x/4,j}$, $u_{1,j}$.

The values of $t_j$, $U_j$, $\langle r^2 \rangle_j$, $u_{N_x/2,j}$, $u_{N_x/4,j}$, $u_{1,j}$ are found in columns 2, 3, 4, 5, 6 and 7 respectively of the file `e.dat`. The `gnuplot` commands

```
gnuplot> ! gfortran diffusionS1.f90 -o d
gnuplot> ! echo "10 100 0.4" | ./d
```

compile and run the program within `gnuplot`. They set $N_x = 10$, $N_t = 100$, $t_f = 0.4$, $\Delta x \approx 0.111$, $\Delta t \approx 4.0404$, $\Delta t/\Delta x^2 \approx 0.327$. The `gnuplot`

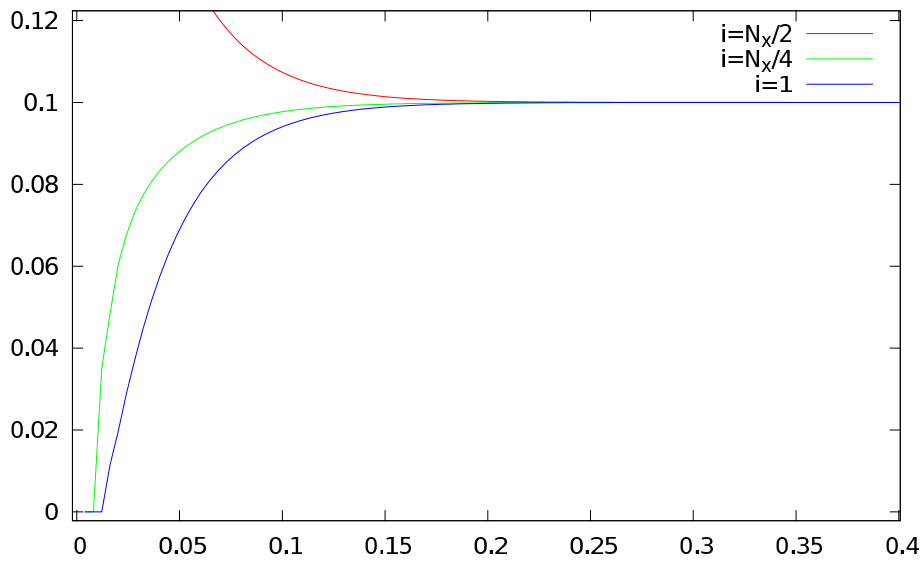



Figure 8.4: The functions $u_{N_x/2,j}$, $u_{N_x/4,j}$, $u_{1,j}$ are given as a function of $t_j$ for $N_x = 10$, $N_t = 100$, $t_f = 0.4$. We observe that for large times they are consistent with uniform diffusion.

commands

```
gnuplot> plot   "e.dat" u 2:5 w l
gnuplot> replot "e.dat" u 2:6 w l
gnuplot> replot "e.dat" u 2:7 w l
```

construct the plot in figure 8.4. We observe that for large times we obtain uniform diffusion.

The relation $U_j = 1$ can be easily confirmed by inspecting the values recorded in the file `e.dat`.

The asymptotic relation $\langle r^2 \rangle_j \sim 2t_j$ can be confirmed with the commands

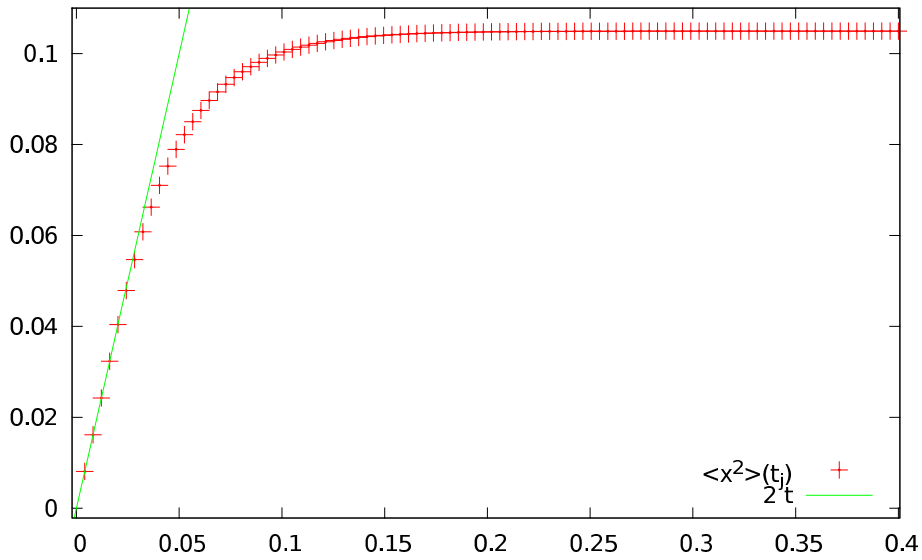



Figure 8.5: The expectation value $\langle r^2 \rangle_j$ as a function of $t_j$ for $N_x = 10$, $N_t = 100$, $t_f = 0.4$. For small values of $t_j$ we obtain $\langle r^2 \rangle_j \approx 2t_j$. The solid line is the straight line $2t$.

```
gnuplot> plot [:][:0.11] "e.dat" u 2:4,2*x
```

which construct the plot in figure 8.5.

Finally we make a plot of the function $u(x,t)$ with the commands

```
gnuplot> ! echo "10 100 0.16" | ./d
gnuplot> set pm3d
gnuplot> splot [0:0.16][0:1][0: 1] "d.dat" w l
gnuplot> splot [0:0.16][0:1][0:.2] "d.dat" w l
```

and the result is shown in figure 8.6.

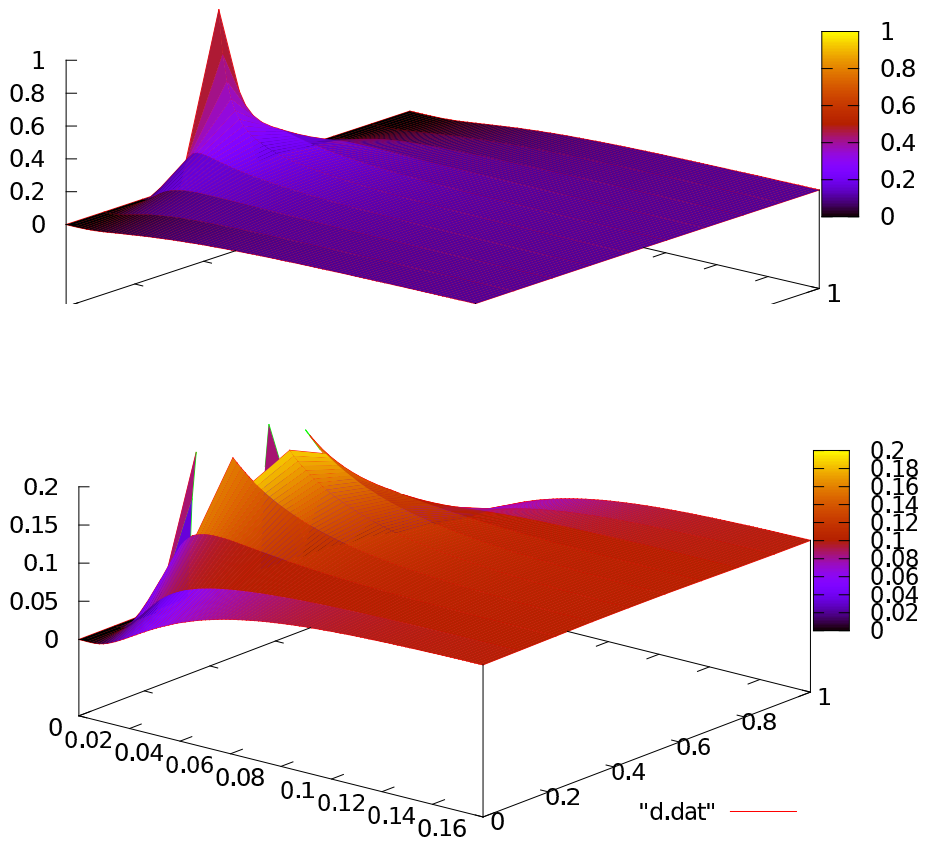



Figure 8.6: The function $u(x,t)$ for $N_x = 10$, $N_t = 100$, $t_f = 0.16$. The second plot differs only in the scale of the $z$ axis so that we can easily see the details of the diffusion away from the point $x_0 \equiv x_{N_x/2} = x_5$.

## 8.8 Problems

8.1 Reproduce the results in figure 8.3.

8.2 The temperature distribution $u(x,t)$ in a thin rod satisfies equation (8.14) together with the boundary conditions (8.13) at the ends $x = 0, 1$. The initial temperature distribution at $t = 0$ is given by the function

$$u(x,0) = \begin{cases} 0.5 & x \in [x_1, x_2] \\ 0.3 & x \notin [x_1, x_2] \end{cases} ,$$

where $x_1 = 0.25$ and $x_2 = 0.75$.

(a) Calculate the temperature distribution $u(x,t_f)$ for $t_f = 0.0001$, $0.001$, $0.01$, $0.05$. Take $N_x = 100$ and $N_t = 1000$. Do the same for $t_f = 0.1$ by choosing appropriate $N_x$ and keeping $N_t = 1000$. Plot the functions $u(x,t_f)$ in the same plot.

(b) Calculate the maximum value of the temperature graphically for $t_f = 0.0001$, $0.001$, $0.01$, $0.05$, $0.1$, $0.15$, $0.25$. Take $N_x = 100$ and choose an appropriate value for the corresponding $N_t$.

(c) Calculate the time at which the temperature of the rod becomes everywhere less than $0.1$.

Hint: Make your program print only the final temperature distribution $u(x,t_f)$.

8.3 The temperature distribution $u(x,t)$ in a thin rod satisfies the equation

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} .$$

The temperature at the ends of the rod is $u(0,t) = u(1,t) = 0$, and when $t = 0$

$$u(x,0) = \begin{cases} 0.5\left[1 - \cos\left(\frac{2\pi x}{b}\right)\right] & 0 \le x < b \\ 0 & b \le x \le 1 \end{cases} .$$

(a) Calculate the temperature distribution $u(x,t_f)$ for $\alpha = 0.5$, $b = 0.09$ and for $t_f = 0.0001$, $0.001$, $0.01$, by taking $N_x = 300$, $N_t = 1000$. Do the same for $t_f = 0.05$ by choosing appropriate $N_x$. Plot the functions $u(x,t_f)$ in the same plot.

(b) Using the same parameters, calculate the time evolution of the values of the temperature distribution at the points $x_1 = 0.05$, $x_2 = 0.50$ and $x_3 = 0.95$ for $0 \le t \le 0.05$. Plot the functions $u(x_{1,2,3},t)$ in the same plot.

(c) Calculate the temperature distribution $u(x, t_f)$ for $b = 0.09$ and $\alpha = 5, 2, 1$ for $t_f = 0.001$. Plot the functions $u(x, t_f)$ in the same plot. Comment on the effect of the parameter $\alpha$ on your results.

8.4 The temperature distribution $u(x, t)$ in a thin rod of length $L$ satisfies equation

$$\frac{\partial u}{\partial t} = D(x)\frac{\partial^2 u}{\partial x^2} - \frac{4}{L}D(x)\frac{\partial u}{\partial x},$$

where $D(x) = a\mathrm{e}^{-4x/L}$ is the $x$-dependent thermal diffusivity. The temperature of the rod at its ends is such that $u(0, t) = u(L, t) = 0$, and at time $t = 0$, the temperature distribution is

$$u(x, 0) = C\mathrm{e}^{-(x-L/2)^2/\sigma^2}.$$

(a) Write a program where the user enters the parameters $L$, $a$, $C$, $\sigma$, $N_x$, $N_t$ and $t_f$ interactively. On exit, the program calculates $u(x, t_f)$ and writes the points $(x_i, u(x_i, t_f))$ in two columns to a file `d.dat`.

(b) Run the program for $L = 4$, $a = 0.2$, $C = 1$, $\sigma = 1/2$, $N_x = 400$, $N_t = 20000$ and calculate $u(x, t_f)$ for $t_f = 0.05, 1.0, 5.0$. Plot the functions $u(x, t_f)$ in the same plot.

(c) Using the same parameters, calculate the time evolution of the temperature distribution at the points $x_1 = 1$ and $x_2 = 2$ for $0 \le t \le 5$. Plot the functions $u(x_{1,2}, t)$ in the same plot.

8.5 Reproduce the results shown in figures 8.4 and 8.5.

# Chapter 9

# The Anharmonic Oscillator

In this chapter we will use matrix methods in order to compute the quantum mechanical energy spectrum of the anharmonic oscillator. This problem cannot be solved exactly and one has to resort to perturbative or other approximation methods. We will approach this problem numerically by representing the Hamiltonian $H$ as a real symmetric matrix in an appropriately chosen basis of the Hilbert space $\mathcal{H}$ of quantum mechanical states. The energy spectrum is obtained from the eigenvalues of this matrix and the numerical problem reduces to that of the diagonalization of a real symmetric matrix. Since the Hamiltonian is represented in $\mathcal{H}$ by an infinite size matrix, we have to restrict ourselves to a finite dimensional subspace $\mathcal{H}_N$ of dimension $N$. In this space the Hamiltonian is represented by an $N \times N$ real symmetric matrix. The eigenvalues of this matrix will be calculated numerically using standard methods and the energy eigenvalues will be obtained in the $N \to \infty$ limit.

For the calculation of the eigenvalues we will use software that is found in the well known library Lapack which contains high quality, freely available, linear algebra software. Part of the goals of this chapter is to show to the reader how to link her programs with software libraries. In order to solve the same problem using Mathematica or Matlab see [37] and [38] respectively.

## 9.1 Introduction

The Hamiltonian of the harmonic oscillator is given by

$$H_0 = \frac{p^2}{2m} + \frac{1}{2} m\omega^2 x^2 \,. \tag{9.1}$$

Define the position and momentum scales $x_0 = \sqrt{\hbar/(m\omega)}$, $p_0 = \sqrt{\hbar m\omega}$ so that we can express the above equation using dimensionless terms:

$$\frac{H_0}{\hbar\omega} = \frac{1}{2}\left(\frac{p}{p_0}\right)^2 + \frac{1}{2}\left(\frac{x}{x_0}\right)^2 . \tag{9.2}$$

If we take the units of energy, distance and momentum to be $\hbar\omega$, $x_0$ and $p_0$, then we obtain

$$H_0 = \frac{1}{2}p^2 + \frac{1}{2}x^2 , \tag{9.3}$$

where $H_0$, $p$ and $x$ are now dimensionless. The operator $H_0$ can be diagonalized with the help of the creation and annihilation operators $a$ and $a^\dagger$, defined by the relations:

$$x = \frac{1}{\sqrt{2}}(a^\dagger + a) \qquad p = \frac{i}{\sqrt{2}}(a^\dagger - a) , \tag{9.4}$$

or

$$a = \frac{1}{\sqrt{2}}(x + ip) \qquad a^\dagger = \frac{1}{\sqrt{2}}(x - ip) , \tag{9.5}$$

which obey the commutation relation

$$[a, a^\dagger] = 1 , \tag{9.6}$$

which leads to

$$H_0 = a^\dagger a + \frac{1}{2} . \tag{9.7}$$

The eigenstates $|n\rangle$, $n = 0, 1, 2, \ldots$ of $H_0$ span the Hilbert space of states $\mathcal{H}$ and satisfy the relations

$$a^\dagger |n\rangle = \sqrt{n+1}\, |n+1\rangle \qquad a\,|n\rangle = \sqrt{n}\,|n-1\rangle \qquad a\,|0\rangle = 0 , \tag{9.8}$$

therefore

$$a^\dagger\, a\,|n\rangle = n\,|n\rangle , \tag{9.9}$$

and

$$H_0\,|n\rangle = E_n\,|n\rangle , \qquad E_n = n + \frac{1}{2} . \tag{9.10}$$

The position representation of the eigenstates $|n\rangle$ is given by the wave-functions:

$$\psi_n(x) = \langle x|n\rangle = \frac{1}{\sqrt{2^n n! \sqrt{\pi}}} \mathrm{e}^{-x^2/2} H_n(x) , \tag{9.11}$$

where $H_n(x)$ are the Hermite polynomials.

From equations (9.4) and (9.8) we obtain

$$x_{nm} = \langle n| x |m\rangle = \frac{1}{\sqrt{2}}\sqrt{m+1}\,\delta_{n,m+1} + \frac{1}{\sqrt{2}}\sqrt{m}\,\delta_{n,m-1} \tag{9.12}$$

$$= \frac{1}{2}\sqrt{n+m+1}\,\delta_{|n-m|,1} \tag{9.13}$$

$$p_{nm} = \langle n| p |m\rangle = -\frac{i}{\sqrt{2}}\sqrt{m+1}\,\delta_{n,m+1} + \frac{i}{\sqrt{2}}\sqrt{m}\,\delta_{n,m-1}\,. \tag{9.14}$$

From the above equations we can easily calculate the Hamiltonian of the *anharmonic* oscillator

$$H(\lambda) = H_0 + \lambda x^4\,. \tag{9.15}$$

The matrix elements of $H$ in this representation are:

$$H_{nm}(\lambda) \equiv \langle n| H(\lambda) |m\rangle = \langle n| H_0 |m\rangle + \lambda\langle n| x^4 |m\rangle \tag{9.16}$$

$$= (n+\frac{1}{2})\delta_{n,m} + \lambda(x^4)_{nm} \tag{9.17}$$

where $(x^4)_{nm}$ can be calculated from equation (9.12):

$$(x^4)_{nm} = \sum_{i_1,i_2,i_3=0}^{\infty} x_{ni_1}x_{i_1i_2}x_{i_2i_3}x_{i_3m}\,. \tag{9.18}$$

This relation computes the matrix elements of the matrix $x^4$ from the matrix product of $x$ with itself.

The problem of the calculation of the energy spectrum has now been reduced to the problem of calculating the eigenvalues of the matrix $H_{nm}$.

## 9.2 Calculation of the Eigenvalues of $H_{nm}(\lambda)$

We start by choosing the dimension $N$ of the subspace $\mathcal{H}_N$ of the Hilbert space of states $\mathcal{H}$. We will restrict ourselves to states within this subspace and we will use the $N$ dimensional representation matrices of $x$, $H_0$ and $H(\lambda)$ in $\mathcal{H}_N$. For example, when $N = 4$ we obtain

$$x = \begin{pmatrix} 0 & \frac{1}{\sqrt{2}} & 0 & 0 \\ \frac{1}{\sqrt{2}} & 0 & 1 & 0 \\ 0 & 1 & 0 & \sqrt{\frac{3}{2}} \\ 0 & 0 & \sqrt{\frac{3}{2}} & 0 \end{pmatrix} \tag{9.19}$$

$$H_0 = \begin{pmatrix} \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{3}{2} & 0 & 0 \\ 0 & 0 & \frac{5}{2} & 0 \\ 0 & 0 & 0 & \frac{7}{2} \end{pmatrix} \tag{9.20}$$

$$H(\lambda) = \begin{pmatrix} \frac{1}{2} + \frac{3\lambda}{4} & 0 & \frac{3\lambda}{\sqrt{2}} & 0 \\ 0 & \frac{3}{2} + \frac{15\lambda}{4} & 0 & 3\sqrt{\frac{3}{2}}\lambda \\ \frac{3\lambda}{\sqrt{2}} & 0 & \frac{5}{2} + \frac{27\lambda}{4} & 0 \\ 0 & 3\sqrt{\frac{3}{2}}\lambda & 0 & \frac{7}{2} + \frac{15\lambda}{4} \end{pmatrix} \tag{9.21}$$

Our goal is to write a program that calculates the eigenvalues $E_n(N, \lambda)$ of the $N \times N$ matrix $H_{nm}(\lambda)$. Instead of reinventing the wheel, we will use ready made routines that calculate eigenvalues and eigenvectors of matrices found in the Lapack library. This library can be found in the high quality numerical software repository `Netlib` and more specifically at `http://www.netlib.org/lapack/`. Documentation can be found at `http://www.netlib.org/lapack/lug/`, but it is also easily accessible online by a Google search or by using the man pages[1].

As inexperienced users we will first look for driver routines that perform a diagonalization process. Since our task is to diagonalize a real symmetric matrix, we pick the subroutine `DSYEV` (D = double precision, SY = symmetric, EV = eigenvalues with optional eigenvectors). If the documentation of the library is installed in our system, we may use the Linux man pages for accessing it:[2]

```
> man dsyev
```

From this page we learn how to use this subroutine:

```
SUBROUTINE DSYEV( JOBZ, UPLO, N, A, LDA, W, WORK, LWORK, INFO )
 CHARACTER      JOBZ, UPLO
 INTEGER        INFO, LDA, LWORK, N
 DOUBLE         PRECISION A( LDA, * ),W( * ), WORK( * )

ARGUMENTS
 JOBZ  (input) CHARACTER*1
```

[1]The library can be easily installed in many Linux distributions. For example in Ubuntu or other Debian like systems you may use the command `apt-get install liblapack3 liblapack-doc liblapack-dev`.

[2]A Google search "dsyev" will easily take you to the same page.

```
      = 'N':  Compute eigenvalues only;
      = 'V':  Compute eigenvalues and eigenvectors.

UPLO  (input) CHARACTER*1
      = 'U':  Upper triangle of A is stored;
      = 'L':  Lower triangle of A is stored.

N     (input) INTEGER
      The order of the matrix A.  N >= 0.

A     (input/output) DOUBLE PRECISION array, dimension (LDA, N↩
   )
      On  entry,  the symmetric matrix A.  If UPLO = 'U', the
      leading N-by-N upper triangular part of A contains the
      upper triangular part  of the matrix A.  If UPLO = 'L',
      the leading N-by-N lower triangular part of A contains
      the lower triangular part of  the matrix A.  On exit, if
      JOBZ = 'V', then if INFO = 0, A contains
      the orthonormal eigenvectors of the matrix A.  If
      JOBZ = 'N', then on exit the lower triangle (if UPLO='L↩
         ')
      or the upper triangle (if UPLO='U') of A, including the
      diagonal, is destroyed.
LDA   (input) INTEGER
      The leading dimension of the array A.  LDA >= max(1,N).

W     (output) DOUBLE PRECISION array, dimension (N)
      If INFO = 0, the eigenvalues in ascending order.

WORK  (workspace/output) DOUBLE PRECISION array, dimension
      (LWORK).
      On exit, if INFO = 0, WORK(1) returns the optimal LWORK.

LWORK (input) INTEGER
      The  length  of  the  array  WORK.  LWORK >= max(1,3*N↩
         -1).
      For optimal efficiency, LWORK >= (NB+2)*N, where NB is
      the  blocksize for DSYTRD returned by ILAENV.

      If  LWORK  = -1, then a workspace query is assumed; the
      routine only calculates the optimal size of  the  WORK
      array,  returns this  value  as the first entry of the
      WORK array, and no error message related to LWORK is
      issued by XERBLA.

INFO  (output) INTEGER
      = 0:  successful exit
      < 0:  if INFO = -i, the i-th argument had an illegal ↩
         value
```

```
> 0:  if INFO = i, the algorithm failed  to  converge;  ↩
    i
off-diagonal  elements  of an intermediate tridiagonal
form did not converge to zero.
```

These originally cryptic pages contain all the necessary information and the reader should familiarize herself with its format. For a quick and dirty use of the routine, all we need to know is the types and usage of its arguments. These are classified as "input", "output" and "working space" variables (some are in more than one classes). Input is the necessary data that the routine needs in order to perform the computation. Output is where the results of the computation are stored. And working space is the memory provided by the user to the routine in order to store intermediate results.

From the information above we learn that the matrix to be diagonalized is `A` which is a rectangular matrix with the number of its rows and columns $\leq N$. The number of rows `LDA` (`LDA=` "leading dimension of A") can be larger than `N` is which case `DSYEV` will diagonalize the upper left `N`×`N` part of the matrix[3]. In our program we define a large matrix `A(LDA,LDA)` and diagonalize a smaller submatrix `A(N,N)`. This way we can study many values of $N$ using the same matrix. The subroutine can be used in two ways:

- If `JOBZ='N'`, it calculates only the eigenvalues of the matrix `A(N,N)` and stores them in the array `W(N)`, sorted in ascending order. We have to be careful because, upon return, the routine destroys the upper (`UPLO='U'`) or lower (`UPLO='L'`) triangular part of `A`. Since `A` is symmetric, only this part is needed by `DSYEV`. If we need to reuse the matrix `A`, we have to make a backup copy before the call to `DSYEV`.

- If `JOBZ='V'`, it calculates both the eigenvalues and the eigenvectors of the matrix `A(N,N)`. The eigenvalues are stored in the array `W(N)` as before, whereas the corresponding eigenvectors in the columns of the matrix `A(N,N)`. In order to use the eigenvectors, we can use a statement like `v = A(1:N,j)` where the array `v(N)` stores the components of the `j`-th eigenvector of the matrix corresponding to the eigenvalue $\lambda_j$. The eigenvectors are normalized to unity, i.e. $\sum_{i=1}^{N}$`v(i)*v(i)`$= 1$. The matrix `A(N,N)` is destroyed after the call

[3]The number `LDA` is necessary because the matrix element `A(i,j)` is found after `i+(LDA-1)*j` memory positions from `A(1,1)`.

to DSYEV and if we need it we have to make a backup copy before the call.

The reader should also familiarize herself with the use of the workspace array WORK. This is memory space given to the routine for all its intermediate calculations. Determining the size of this array needs some care. This is given by LWORK and if performance is an issue the reader should read the documentation carefully for its optimal determination. We will make the simple choice LWORK=3*LDA-1. The variable INFO is used as a flag which informs the user whether the calculation was successful, in which case its value is set to 0. In our case, if INFO takes a non zero value, the program will abort the calculation.

Before using the program in a complicated calculation, it is necessary to test its use in a simple, easily controlled problem. We will familiarize ourselves with the use of DSYEV by writing the following program:

```
program test_evs
 implicit none
 integer, parameter :: P     = 100 ! P= LDA
 integer, parameter :: LWORK = 3*P-1
 real(8) :: A(P,P),W(P),WORK(LWORK)
 integer :: N  ! DSYEV diagonalizes A(N,N)
 integer :: i,j
 integer :: LDA,INFO
 character(1) :: JOBZ,UPLO
!Define the **symmetric** matrix to be diagonalized
!The subroutine uses the upper triangular part  (UPLO='U')
!therefore the lower triangular part needs not to be defined
 N=4
 A(1,1)=-7.7;
 A(1,2)= 2.1;A(2,2)= 8.3;
 A(1,3)=-3.7;A(2,3)=-16.;A(3,3)=-12.
 A(1,4)= 4.4;A(2,4)= 4.6;A(3,4)=-1.04;A(4,4)=-3.7
!We print the matrix A before calling DSYEV since it is
!destroyed after the call.
 do i=1,N
  do j=i,N
   print *,'A( ',i,' , ',j,' )=',A(i,j)
  enddo
 enddo
!We ask for eigenvalues AND eigenvectors (JOBZ='V')
 JOBZ='V'; UPLO='U'
 print *,'COMPUTING WITH DSYEV:'
 LDA=P                          !notice that LDA-> P>N !!
 call DSYEV(JOBZ,UPLO,N,A,LDA,W,WORK,LWORK,INFO)
 print *,'DSYEV: DONE. CHECKING NOW:'
```

```
!If INFO is nonzero, then there is an error:
 if(INFO .ne. 0)then
  print *,'DSYEV FAILED. INFO= ',INFO
  stop
 endif
!Print results: W(I) has the eigenvalues:
 print *,'DSYEV: DONE.:'
 print *,'EIGENVALUES OF MATRIX:'
 do i=1,N
  print *,'LAMBDA(',i,')=',W(i)
 enddo
!Eigenvectors are in stored in the columns of A:
 print *,'EIGENVECTORS OF MATRIX'
 do J=1,N
  print *,'EIGENVECTOR ',j,' FOR EIGENVALUE ',W(j)
  do i=1,N
   print *,'V_',j,'(',i,')= ',A(i,j)
  enddo
 enddo
end program test_evs
```

The next step is to compile and link the program. In order to link the program to Lapack we have to instruct the *linker* `ld` where to find the libraries Lapack and BLAS[4] and link them to our program. A library contains compiled software in archives of *object* files. The convention for their names in a Unix environment is to start with the string "`lib`" followed by the name of the library and a `.a` or `.so` extension. For example, in our case the files we are interested in have the names `lib`**lapack**`.so` and `lib`**blas**`.so` which can be searched in the file system by the commands:

```
> locate libblas
> locate liblapack
```

In order to see the files that they contain we give the commands[5]:

```
> ar -t /usr/lib/libblas.so
> ar -t /usr/lib/liblapack.so
```

In the commands shown above you may have to substitute `/usr/lib` with the path appropriate for your system. If the program is in the file `test.f90`, the compilation/linking command is:

[4]The library BLAS contains the basic linear algebra subroutines used by Lapack. In some versions of the library, one has to only link to Lapack ignoring the link BLAS but in some other version, linking to BLAS is necessary.

[5]If the `.so` files don't exist in your system, try `ar -t /usr/lib/libblas.a` etc.

```
> gfortran test.f90 -o test -L/usr/lib -llapack -lblas
```

The option `-L/usr/lib` instructs the linker to look for libraries in the `/usr/lib` directory[6], whereas the options `-llapack -lblas` instructs the linker to look for any *unresolved symbols* (i.e. names of external functions and subroutines not coded in our program) *first* in the library `lib`**lapack**`.a` and *then* in the library `lib`**blas**`.a`.

The command shown above produces the executable file `test` which, when run, produces the result:

```
EIGENVALUES OF MATRIX:
LAMBDA( 1)= -21.4119907
LAMBDA( 2)= -9.93394359
LAMBDA( 3)= -2.55765591
LAMBDA( 4)=  18.8035905
EIGENVECTORS OF MATRIX
EIGENVECTOR  1 FOR EIGENVALUE  -21.4119907
V_ 1( 1)=  -0.197845668
V_ 1( 2)=  -0.464798676
V_ 1( 3)=  -0.854691009
V_ 1( 4)=   0.119676904
EIGENVECTOR  2 FOR EIGENVALUE  -9.93394359
V_ 2( 1)=   0.824412399
V_ 2( 2)=  -0.132429396
V_ 2( 3)=  -0.191076519
V_ 2( 4)=  -0.516039161
EIGENVECTOR  3 FOR EIGENVALUE  -2.55765591
V_ 3( 1)=   0.502684215
V_ 3( 2)=  -0.247784372
V_ 3( 3)=   0.132853329
V_ 3( 4)=   0.817472616
EIGENVECTOR  4 FOR EIGENVALUE   18.8035905
V_ 4( 1)=   0.168848655
V_ 4( 2)=   0.839659187
V_ 4( 3)=  -0.464050682
V_ 4( 4)=   0.226096318
```

We are now ready to tackle the problem of computing the energy spectrum of the anharmonic oscillator. The main program contains the user interface where the basic parameters for the calculation are read from the `stdin`. The user can specify the dimension `DIM` $\equiv N$ of $\mathcal{H}_N$ and the coupling constant $\lambda$. Then the program computes the eigenvalues $E_n(N, \lambda)$

[6]This is not necessary in our case, since `/usr/lib` is in the *path* that `ld` searches anyway. This option is useful for libraries located in non conventional paths.

of the $N \times N$ matrix $H_{nm}(\lambda)$, which represents the action of the operator $H(\lambda)$ in the $\{\,|n\rangle\}_{n=0,1,\ldots,N-1}$ representation in $\mathcal{H}_N$. The tasks are allocated to the subroutines `calculate_X4`, `calculate_evs` and `calculate_H`. The subroutine `calculate_X4` calculates the $N \times N$ matrix $(x^4)_{nm}$. First, the matrix $x_{nm}$ is calculated and then $(x^4)_{nm}$ is obtained by computing its fourth power. The matrix $(x^4)_{nm}$ can also be calculated analytically and this is left as an exercise to the reader. The subroutine `calculate_H` calculates the matrix $H_{nm}(\lambda)$ using the result for $(x^4)_{nm}$ given by `calculate_X4`. Finally the eigenvalues are calculated in the subroutine `calculate_evs` by a call to `DSYEV`, which are returned to the main program for printing to the `stdout`. The program is listed below and can be found in the file `anharmonic.f90`:

```
!=========================================================
program anharmonic_elevels
!=========================================================
 implicit none
 integer,parameter        :: P     = 1000
 integer,parameter        :: LWORK = 3*P-1
 integer                  :: DIM
 real(8),dimension(P,P)   :: H,X,X4 !Hamiltionian+Position Ops
 real(8),dimension(P)     :: E      !energy eigenvalues
 real(8),dimension(LWORK):: WORK
 real(8)                  :: lambda
 integer                  :: i

 print *,'# Enter Hilbert Space dimension:'
 read  *,DIM
 print *,'# Enter lambda:'
 read  *,lambda
 print *,'# lambda= ',lambda
!Print Message:
 print *,'# ##############################################'
 print *,'# Energy spectrum of anharmonic oscillator'
 print *,'# using matrix methods.'
 print *,'# Hilbert Space Dimension DIM = ',DIM
 print *,'# lambda coupling = ',lambda
 print *,'# ##############################################'
 print *,'# Outpout: DIM lambda E_0 E_1 .... E_{N-1}'
 print *,'# ----------------------------------------------'

!Calculate X^4 operator:
 call calculate_X4(X,X4,DIM)
!Calculate eigenvalues:
 call calculate_evs(H,X4,E,WORK,lambda,DIM)
 write(6,100)'EV ',DIM,lambda,(E(i),i=1,DIM)
```

```
100 FORMAT(A3,I8,20000G25.15)
end program anharmonic_elevels
!=========================================================
subroutine calculate_evs(H,X4,E,WORK,lambda,DIM)
!=========================================================
 implicit none
 integer,parameter          :: P     = 1000
 integer,parameter          :: LWORK = 3*P-1
 real(8),dimension(P,P)     :: H,X4
 real(8),dimension(P)       :: E
 real(8),dimension(LWORK)   :: WORK
 integer                    :: DIM
 real(8)                    :: lambda
 character(1)               :: JOBZ,UPLO
 integer                    :: LDA,INFO,i,j

 call calculate_H(H,X4,lambda,DIM)
 JOBZ='V';UPLO='U'
 call DSYEV(JOBZ,UPLO,DIM,H,P,E,WORK,LWORK,INFO)
 print *,'# ********************** EVEC ******************'
 do j=1,DIM
  write(6,101)'# EVEC ',lambda,(H(i,j), i=1,DIM)
 enddo
 print *,'# ********************** EVEC ******************'
101 FORMAT(A7,F15.3,20000G14.6)
!If INFO is nonzero then we have an error
 if(INFO .ne. 0)then
  print *,'dsyev failed. INFO= ',INFO
  stop
 endif

end subroutine calculate_evs
!=========================================================
subroutine calculate_H(H,X4,lambda,DIM)
!=========================================================
 implicit none
 integer,parameter        :: P = 1000
 real(8),dimension(P,P)   :: H,X4
 integer                  :: DIM
 real(8)                  :: lambda
 integer                  :: i,j

 do j=1,DIM
  do i=1,DIM
   H(i,j)=lambda*X4(i,j)
  enddo
  H(j,j) = H(j,j) + DBLE(j) - 0.5D0 !E_n=n+1/2,n=j-1=>E_n=j-1/2
 enddo
```

```
 print *,'# ********************** H ******************'
 do j=1,DIM
  write(6,102)'# HH ',(H(i,j), i=1,DIM)
 enddo
 print *,'# ********************** H ******************'

102 FORMAT(A5,20000G20.6)
end subroutine calculate_H
!=========================================================
subroutine  calculate_X4(X,X4,DIM)
!=========================================================
 implicit none
 integer,parameter      :: P=1000
 real(8),dimension(P,P):: X,X4,X2
 integer                :: DIM
 integer                :: i,j,m,n
 real(8),parameter      :: isqrt2=1.0D0/sqrt(2.0D0)
!Compute the position operator:
 X = 0.0D0
!Compute the nonzero elements
 do i=1,DIM
  n=i-1 !indices 0,...,DIM-1
! The delta_{n,m+1} term, i.e. m=n-1
  m=n-1 !the energy level n -> i=n+1, m-> j=m+1
  j=m+1
  if(j.ge.1  ) X(i,j)=isqrt2*sqrt(DBLE(m+1))
! The delta_{n,m-1} term, i.e. m=n+1
  m=n+1
  j=m+1
  if(j.le.DIM) X(i,j)=isqrt2*sqrt(DBLE(m))
 enddo
!Compute the Hamiltonian operator:
!Start with the X^4 operator:
 X2 = MATMUL(X ,X ) !first X2,then X4:
 X4 = MATMUL(X2,X2)
end subroutine calculate_X4
```

## 9.3 Results

Compiling and running the program can be done with the commands:

```
> gfortran -O2 anharmonic.f90 -o an -llapack -lblas
> ./an
 # Enter Hilbert Space dimension:
4
 # Enter lambda:
```

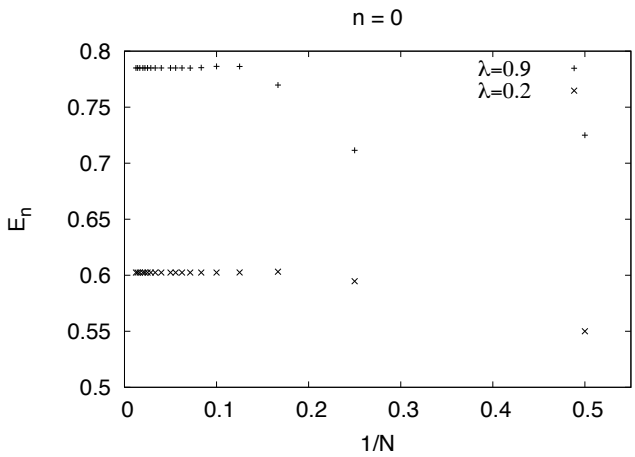



Figure 9.1: The ground state energy $E_0(\lambda)$ for $\lambda = 0.2, 0.9$ is calculated in the large $N$ limit of the eigenvalues $E_0(N, \lambda)$. Convergence is satisfactory for relatively small values of $N$ and it is slightly faster for $\lambda = 0.2$ than it is for $\lambda = 0.9$.

```
0.0
.....
# ********************** H ******************
# HH            0.50    0.00   0.00    0.00
# HH            0.00    1.50   0.00    0.00
# HH            0.00    0.00   2.50    0.00
# HH            0.00    0.00   0.00    3.50
# ********************** H ******************
# ********************** EVEC ***************
# EVEC   0.000    1.00    0.00   0.00    0.00
# EVEC   0.000    0.00    1.00   0.00    0.00
# EVEC   0.000    0.00    0.00   1.00    0.00
# EVEC   0.000    0.00    0.00   0.00    1.00
# ********************** EVEC ***************
EV  4   0.000    0.50    1.50   2.50    3.50
```

In the above program we used $N = 4$ and $\lambda = 0$. The $\lambda = 0$ choice leads us to the simple harmonic oscillator and we obtain the expected solutions: $H_{nm} = (n + 1/2)\delta_{n,m}$, $E_n = (n + 1/2)$ and the eigenstates (eigenvectors of $H_{nm}$) $|n\rangle_{\lambda=0} = |n\rangle = \sum_{m=0}^{3} \delta_{n,m} |m\rangle$. Similar results can be obtained for larger $N$.

For non zero values of $\lambda$, the finite $N$ calculation contains systematic errors from neglecting all the matrix elements $H_{nm}(\lambda)$ for $n \geq N$ or $m \geq N$. Our program calculates the eigenvalues $E_n(N, \lambda)$ of the finite matrix $H_{nm}(\lambda)$, $m, n = 0, \ldots, N - 1$ and one expects that

$$E_n(\lambda) = \lim_{N \to \infty} E_n(N, \lambda)\,, \tag{9.22}$$

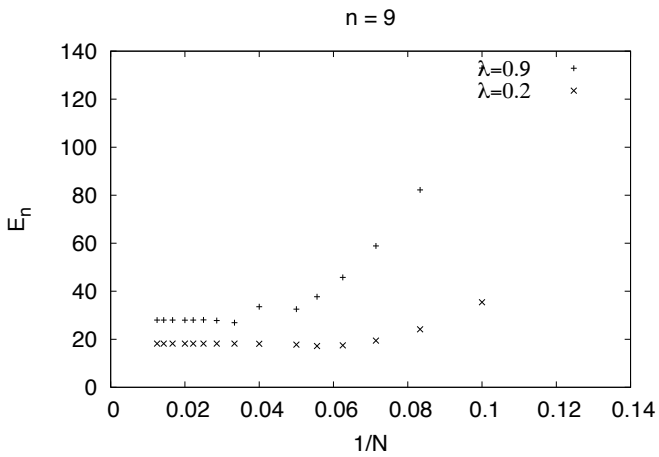



Figure 9.2: The 9th excited state $E_9(\lambda)$ for $\lambda = 0.2, 0.9$ is given by the large $N$ limit of the eigenvalues $E_9(N, \lambda)$.

where

$$H(\lambda)\,|n\rangle_\lambda = E_n(\lambda)\,|n\rangle_\lambda\,, \tag{9.23}$$

is the true $n$-th level eigenvalue of the Hamiltonian $H(\lambda)$. In practice the limit 9.22 for given $\lambda$ and $n$ is calculated by computing $E_n(N, \lambda)$ numerically for increasing values of $N$. If convergence to a desired level of accuracy is achieved for the accessible values of $N$, then the approached limit is taken as an approximation to $E_n(\lambda)$. This process is shown graphically in figures 9.1-9.3 for $\lambda = 0.2, 0.9$. Convergence is satisfactory for quite small $N$ for $n = 0, 9$ but larger values of $N$ are needed for $n = 20$. Increasing the value of $n$ for fixed $\lambda$ makes the use of larger values of $N$ necessary. Similarly for a given energy level $n$, increasing $\lambda$ also makes the use of larger values of $N$ necessary. A session that computes this limit for the ground level energy $E_0(\lambda = 0.9)$ is shown below[7]:

```
> tcsh
> gfortran -O2 anharmonic.f90 -llapack -lblas -o an
> foreach N (4 8 12 16 24 32)
foreach? (echo $N;echo 0.9) |./an >> data
foreach? end
> grep ^EV data | awk '{print $2,$4}'
4  0.711467845686790
8  0.786328966767866
12 0.785237674919165
```

[7]The `foreach` loop construct is special to the `tcsh` shell. This is why an explicit `tcsh` command is shown. For other shells use their corresponding syntax.

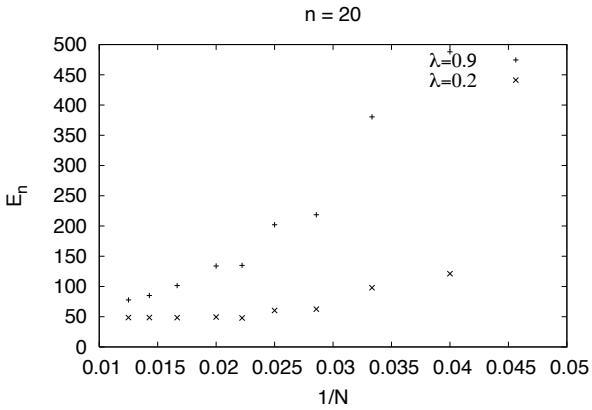



Figure 9.3: The 20th excited state $E_{20}(\lambda)$ for $\lambda = 0.2, 0.9$ is given by the large $N$ limit of the eigenvalues $E_{20}(N, \lambda)$. Convergence has not been achieved for the displayed values of $N \leq 80$.

```
16  0.784964461939594
24  0.785032515135677
32  0.785031492177730
> gnuplot
gnuplot> plot "<grep ^EV data | awk '{print 1/$2,$4}'"
```

Further automation of this process can be found in the shell script file `anharmonic.csh` in the accompanying software. We note the large $N$ convergence of $E_0(N, 0.9)$ and that we can take $E_0(0.9) \approx 0.78503$. For higher accuracy, a computation using larger $N$ will be necessary.

We can also compute the expectation values $\langle A \rangle_n(\lambda)$ of an operator $A = A(p, q)$ when the anharmonic oscillator is in a state $|n\rangle_\lambda$:

$$\langle A \rangle_n(\lambda) = {}_\lambda\langle n| \, A \, |n\rangle_\lambda \, . \tag{9.24}$$

In practice, the expectation value will be computed from the limit

$$\langle A \rangle_n(\lambda) = \lim_{N\to\infty} \langle A \rangle_n(N, \lambda) \equiv \lim_{N\to\infty} {}_{N,\lambda}\langle n| \, A \, |n\rangle_{N,\lambda} \, , \tag{9.25}$$

where $|n\rangle_{N,\lambda}$ are the eigenvectors of the finite $N \times N$ matrix $H_{nm}(\lambda)$ computed numerically by `DSYEV`. These are determined by their components $c_m(N, \lambda)$, where

$$|n\rangle_{N,\lambda} = \sum_{m=0}^{N-1} c_m(N, \lambda) \, |m\rangle \, , \tag{9.26}$$

which are stored in the columns of the array `H` after the call to `DSYEV`:

$$c_m(N, \lambda) = \mathtt{H}(m + 1, n + 1) \, . \tag{9.27}$$

Substituting equation (9.26) to (9.24) we obtain

$$\langle A\rangle_n(\lambda) = \sum_{m,m'=0}^{N-1} c_m^*(N,\lambda)c_{m'}(N,\lambda)A_{mm'}\,, \tag{9.28}$$

and we can use (9.27) for the computation of the sum.

As an application, consider the expectation values of the operators $x^2$, $x^4$ and $p^2$. Taking into account that $\langle x\rangle_n = \langle p\rangle_n = 0$, we obtain the uncertainties $\Delta x_n \equiv \sqrt{\langle x^2\rangle_n - \langle x\rangle_n^2} = \sqrt{\langle x^2\rangle_n}$ and $\Delta p_n = \sqrt{\langle p^2\rangle_n}$. Their product should satisfy Heisenberg's uncertainty relation $\Delta x_n \cdot \Delta p_n \gtrsim 1/2$. The results are shown in table 9.1 and in figures 9.4-9.5. The calculation is left as an exercise to the reader.

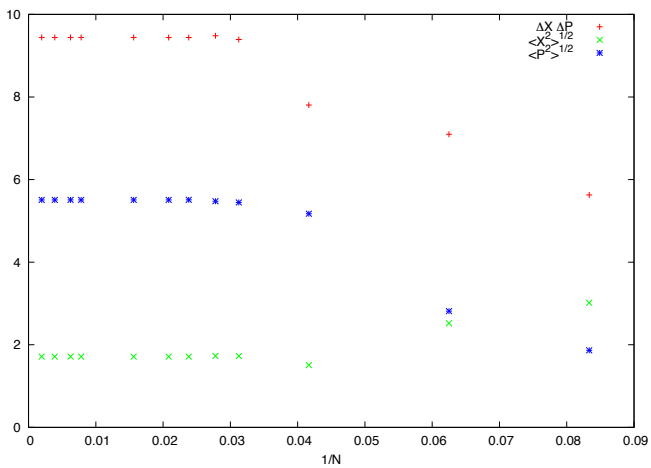



Figure 9.4: The expectation values $\langle x^2\rangle_n^{1/2}(\lambda)$, $\langle p^2\rangle_n^{1/2}(\lambda)$ and the product of uncertainties $\Delta x_n \cdot \Delta p_n$ for $n = 9$ and $\lambda = 0.5$ calculated from the large $N$ limits of $\langle x^2\rangle_n^{1/2}(N,\lambda)$, $\langle p^2\rangle_n^{1/2}(N,\lambda)$.

The physics of the anharmonic oscillator can be better understood by studying the large $\lambda$ limit. As shown in figure 9.5, the term $\lambda x^4$ dominates in this limit and the expectation value $\langle x^2\rangle_n(\lambda)$ decreases. This means that states that confine the oscillator to a smaller range of $x$ are favored. This, using the uncertainty principle, implies that the typical momentum of the oscillator also increases in magnitude. This is confirmed in figure 9.5 where we observe the expectation value $\langle p^2\rangle_n(\lambda)$ to increase with $\lambda$. In order to understand quantitatively these competing effects we will use a scaling argument due to Symanzik. We redefine $x \to \lambda^{-1/6}x$, $p \to \lambda^{1/6}p$ in the Hamiltonian $H(\lambda) = p^2/2 + x^2/2 + \lambda x^4$ and for large enough $\lambda$ we

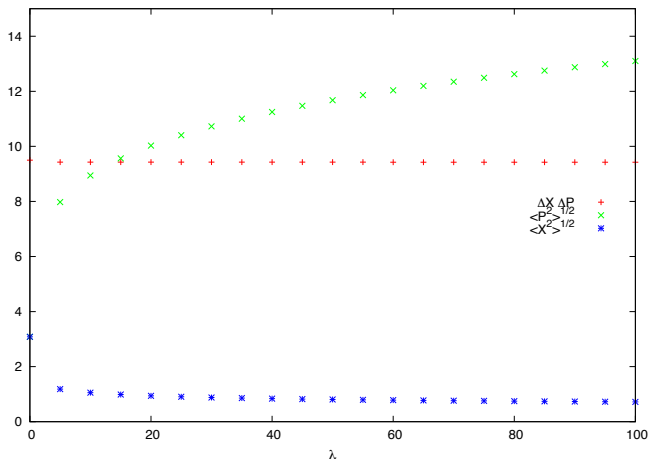



Figure 9.5: The expectation values $\langle x^2\rangle_n^{1/2}(\lambda)$, $\langle p^2\rangle_n^{1/2}(\lambda)$ and the product of uncertainties $\Delta x_n \cdot \Delta p_n$ for $n = 9$.

obtain[8] the asymptotic behavior

$$H(\lambda) \sim \lambda^{1/3}\, h(1)\,, \qquad \lambda \to \infty\,, \tag{9.29}$$

where $h(\lambda) = p^2/2 + \lambda x^4$ is the Hamiltonian of the anharmonic "oscillator" with $\omega = 0$. Since the operator $h(1)$ is independent of $\lambda$, the energy spectrum will have the asymptotic behavior

$$E_n(\lambda) \sim C_n \lambda^{1/3}\,, \qquad \lambda \to \infty\,. \tag{9.30}$$

In reference [39] it is shown that for $\lambda > 100$ we have that

$$E_0(\lambda) = \lambda^{1/3}\left(0.667\,986\,259\,18 + 0.143\,67\lambda^{-2/3} - 0.0088\lambda^{-4/3} + \ldots\right)\,, \tag{9.31}$$

with an accuracy better than one part in $10^6$. For large values of $n$, the authors obtain the asymptotic behavior

$$E_n(\lambda) \sim C\lambda^{1/3}\left(n + \frac{1}{2}\right)^{4/3}\,, \qquad \lambda \to \infty\,, n \to \infty\,, \tag{9.32}$$

where $C = 3^{4/3}\pi^2/\Gamma(1/4)^{8/3} \approx 1.376\,507\,40$. This relation is tested in figure 9.6 where we observe good agreement with our calculations.

[8]For $x \to \lambda^{-1/6}x$, $H \to \lambda^{1/3}(p^2/2 + \lambda^{-2/3}x^2/2 + x^4)$, therefore in the limit $\lambda \to \infty$ the second term vanishes and we obtain equation (9.29).

| | $\lambda = 0.5$ | | | $\lambda = 2.0$ | | |
|---|---|---|---|---|---|---|
| $n$ | $\langle x^2 \rangle$ | $\langle p^2 \rangle$ | $\Delta x \cdot \Delta p$ | $\langle x^2 \rangle$ | $\langle p^2 \rangle$ | $\Delta x \cdot \Delta p$ |
| 0 | 0.305814 | 0.826297 | 0.502686 | 0.21223 | 1.19801 | 0.504236 |
| 1 | 0.801251 | 2.83212 | 1.5064 | 0.540792 | 4.21023 | 1.50893 |
| 2 | 1.15544 | 5.38489 | 2.49438 | 0.761156 | 8.15146 | 2.49089 |
| 3 | 1.46752 | 8.28203 | 3.48627 | 0.958233 | 12.6504 | 3.48166 |
| 4 | 1.75094 | 11.4547 | 4.47845 | 1.13698 | 17.596 | 4.47285 |
| 5 | 2.01407 | 14.8603 | 5.47079 | 1.30291 | 22.9179 | 5.46443 |
| 6 | 2.2617 | 18.4697 | 6.4632 | 1.45905 | 28.5683 | 6.45619 |
| 7 | 2.49696 | 22.2616 | 7.45562 | 1.60735 | 34.5124 | 7.44805 |
| 8 | 2.72198 | 26.2196 | 8.44804 | 1.74919 | 40.7234 | 8.43998 |
| 9 | 2.93836 | 30.3306 | 9.44045 | 1.88558 | 47.1801 | 9.43194 |

Table 9.1: The expectation values $\langle x^2 \rangle$, $\langle p^2 \rangle$, $\Delta x \cdot \Delta p$ for the anharmonic oscillator for the states $|n\rangle$, $n = 0, \ldots, 9$. We observe a decrease of $\Delta x = \sqrt{\langle x^2 \rangle}$ and an increase of $\Delta p = \sqrt{\langle p^2 \rangle}$ as $\lambda$ is increased. The product $\Delta x \cdot \Delta p$ seems to remain very close to the values $(n + 1/2)$ of the harmonic oscillator for both values of $\lambda$.

## 9.4 The Double Well Potential

We can also use matrix methods in order to calculate the energy spectrum of a particle in a double well potential given by the Hamiltonian:

$$H = \frac{p^2}{2} - \frac{x^2}{2} + \lambda \frac{x^4}{4} \,. \tag{9.33}$$

The equilibrium points of the classical motion are located at the minima:

$$x_0 = \pm \frac{1}{\sqrt{\lambda}}, V_{min} = -\frac{1}{4\lambda} \,. \tag{9.34}$$

When the well is very deep, then for the lowest energy levels the potential can be well approximated by that of a harmonic oscillator with angular frequency $\omega^2 = V''(x_0)$, therefore

$$E_{min} \approx V_{min} + \frac{1}{2}\omega \,. \tag{9.35}$$

In this case the tunneling effect is very weak and the energy levels are arranged in almost degenerate pairs. The corresponding eigenstates are symmetric and antisymmetric linear combinations of states localized near the left and right minima of the potential. For example, for the two lowest

Table 9.2: Numerical calculation of the energy levels of the anharmonic oscillator given in reference [39].

| $\lambda$ | $E_0$ | $E_1$ | $E_2$ | $E_3$ | $E_4$ |
|---|---|---|---|---|---|
| 0.002 | 0.501 489 66 | 1.507 419 39 | 2.519 202 12 | 3.536 744 13 | 4.559 955 56 |
| 0.006 | 0.504 409 71 | 1.521 805 65 | 2.555 972 30 | 3.606 186 33 | 4.671 800 37 |
| 0.01 | 0.507 256 20 | 1.535 648 28 | 2.590 845 80 | 3.671 094 94 | 4.774 913 12 |
| 0.05 | 0.532 642 75 | 1.653 436 01 | 2.873 979 63 | 4.176 338 91 | 5.549 297 81 |
| 0.1 | 0.559 146 33 | 1.769 502 64 | 3.138 624 31 | 4.628 882 81 | 6.220 300 90 |
| 0.3 | 0.637 991 78 | 2.094 641 99 | 3.844 782 65 | 5.796 573 63 | 7.911 752 73 |
| 0.5 | 0.696 175 82 | 2.324 406 35 | 4.327 524 98 | 6.578 401 95 | 9.028 778 72 |
| 0.7 | 0.743 903 50 | 2.509 228 10 | 4.710 328 10 | 7.193 265 28 | 9.902 610 70 |
| 1 | 0.803 770 65 | 2.737 892 27 | 5.179 291 69 | 7.942 403 99 | 10.963 5831 |
| 2 | 0.951 568 47 | 3.292 867 82 | 6.303 880 57 | 9.727 323 19 | 13.481 2759 |
| 50 | 2.499 708 77 | 8.915 096 36 | 17.436 9921 | 27.192 6458 | 37.938 5022 |
| 200 | 3.930 931 34 | 14.059 2268 | 27.551 4347 | 43.005 2709 | 60.033 9933 |
| 1000 | 3.694 220 85 | 23.972 2061 | 47.017 3387 | 73.419 1140 | 102.516 157 |
| 8000 | 13.366 9076 | 47.890 7687 | 93.960 6046 | 146.745 512 | 204.922 711 |
| 20000 | 18.137 2291 | 64.986 6757 | 127.508 839 | 199.145 124 | 278.100 238 |
| $\lambda$ | $E_5$ | $E_6$ | $E_7$ | $E_8$ | |
| 0.002 | 5.588 750 05 | 6.623 044 60 | 7.662 759 33 | 8.707 817 30 | |
| 0.006 | 5.752 230 87 | 6.846 948 47 | 7.955 470 29 | 9.077 353 66 | |
| 0.01 | 5.901 026 67 | 7.048 326 88 | 8.215 837 81 | 9.402 692 31 | |
| 0.05 | 6.984 963 10 | 8.477 397 34 | 10.021 9318 | 11.614 7761 | |
| 0.1 | 7.899 767 23 | 9.657 839 99 | 11.487 3156 | 13.378 9698 | |
| 0.3 | 10.166 4889 | 12.544 2587 | 15.032 7713 | 17.622 4482 | |
| 0.5 | 11.648 7207 | 14.417 6692 | 17.320 4242 | 20.345 1931 | |
| 0.7 | 12.803 9297 | 15.873 6836 | 19.094 5183 | 22.452 9996 | |
| 1 | 14.203 1394 | 17.634 0492 | 21.236 4362 | 24.994 9457 | |
| 2 | 17.514 1324 | 21.790 9564 | 26.286 1250 | 30.979 8830 | |
| 50 | 49.516 4187 | 61.820 3488 | 74.772 8290 | 88.314 3280 | |
| 200 | 78.385 6232 | 97.891 3315 | 118.427 830 | 139.900 400 | |
| 1000 | 133.876 891 | 167.212 258 | 202.311 200 | 239.011 580 | |
| 8000 | 267.628 498 | 334.284 478 | 404.468 350 | 477.855 700 | |
| 20000 | 363.201 843 | 453.664 875 | 548.916 140 | 648.515 330 | |

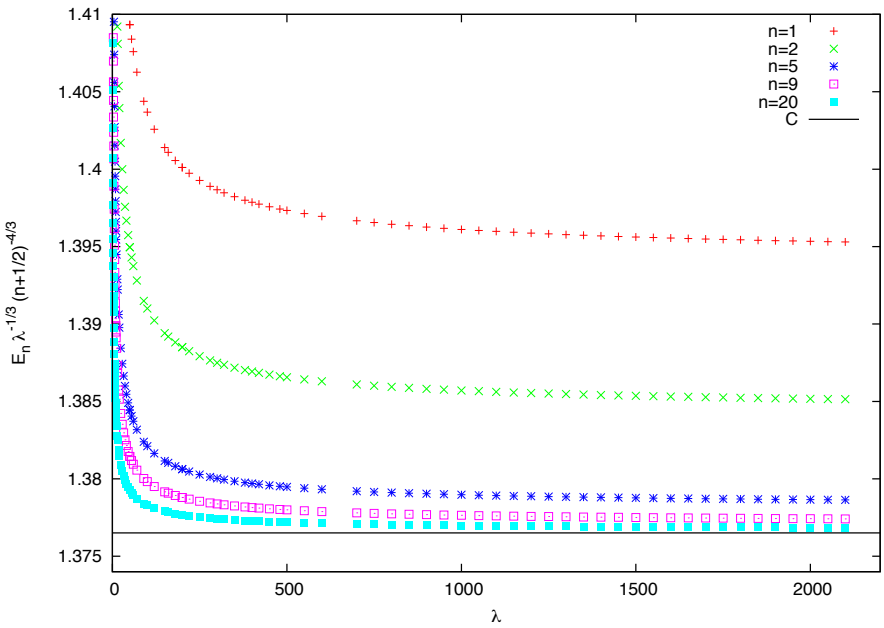



Figure 9.6: Test of the asymptotic relation (9.32). The vertical axis is $E_n\lambda^{-1/3}(n+1/2)^{-4/3}$ where for large enough $n$ and $\lambda$ should approach the value $C = 3^{4/3}\pi^2/\Gamma(1/4)^{8/3} \approx 1.376\,507\,40$ (horizontal line).

energy levels we expect that

$$E_{0,1} \approx E_{min} \pm \frac{\Delta}{2}\,, \tag{9.36}$$

where $\Delta \ll |E_{min}|$ and

$$|0\rangle_\lambda \approx \frac{|+\rangle + |-\rangle}{\sqrt{2}}\,, \qquad |1\rangle_\lambda \approx \frac{|+\rangle - |-\rangle}{\sqrt{2}}\,, \tag{9.37}$$

where the states $|+\rangle$ and $|-\rangle$ are localized to the left and right well of the potential respectively (see also figure 10.4 of chapter 10).

We will use equations (9.12) in order to calculate the Hamiltonian (9.33). We need to make very small modifications to the code in the file `anharmonic.f90`. We will only add a routine that calculates the matrices $p_{nm}$. The resulting program can be found in the file `doublewell.f90`:

```
!=========================================================
program doublewell_elevels
!=========================================================
! H       : Hamiltonian operator H0+(lambda/4)*X^4
```

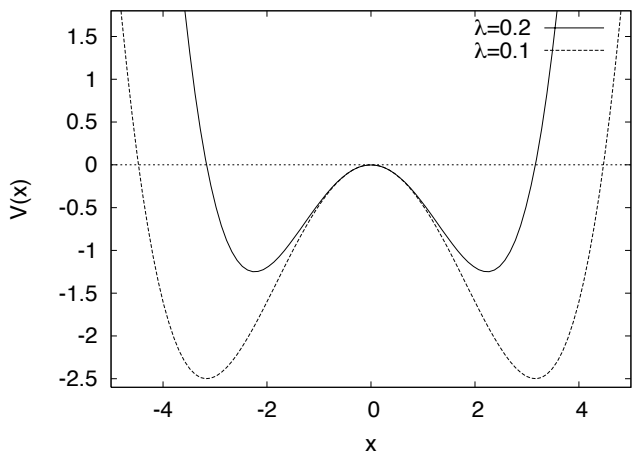



Figure 9.7: The potential energy $V(x)$ for the double well potential for $\lambda = 0.1, 0.2$.

```
! H0      : Hamiltonian H0=1/2 P^2-1/2 X^2
! X,X2,X4: Position operator and its powers
! iP      : i P operator
! P2      : P^2 = -(iP)(iP) operator
! E       : Energy eigenvalues
! WORK    : Workspace for lapack routine DSYEV
!=========================================================
 implicit none
 integer,parameter          :: P=1000
 integer,parameter          :: LWORK=3*P-1
 real(8),dimension(P,P)     :: H,H0,X,X4,X2,iP,P2
 real(8),dimension(P)       :: E
 real(8),dimension(LWORK)   :: WORK
 real(8)                    :: lambda,lambda0,lambdaf,dlambda
 integer                    :: DIM0,DIMF,dDIM,DIM
 integer                    :: i

!Minimum and maximum values of Hilbert space dimensions:
 print *,'Enter Hilbert Space dimensions (DIM0,DIMF,DDIM):'
 read  *,DIM0,DIMF,DDIM
!Minimum and maximum values of lambda (step dlambda):
 print *,'Enter lambda0,lambdaf,dlambda:'
 read  *,lambda0,lambdaf,dlambda
 print *,'lambda0= ',lambda0
!Print Message:
 print *,'# ##############################################'
 print *,'# Energy levels of double well potential'
 print *,'# using matrix methods.'
 print *,'# Hilbert Space Dimensions = ',DIM0,' - ',DIMF,&
```

```
      ' step= ',dDIM
 print *,'# lambda coupling = ',lambda0,' - ',lambdaf,&
      ' step= ',dlambda
 print *,'# ##############################################'
 print *,'# Outpout: DIM lambda E_0 E_1 .... E_{N-1}'
 print *,'# ----------------------------------------------'

 do DIM=DIM0,DIMF,dDIM

  call calculate_operators(X,X2,X4,iP,P2,H0,DIM)

  lambda = lambda0
  do while (lambda .le. lambdaf )
   call calculate_evs(H,H0,X4,E,WORK,lambda,DIM)
   write(6,100)'EV ',DIM,lambda,(E(i),i=1,DIM)
   lambda = lambda+dlambda
  enddo
 enddo
100 FORMAT(A3,I5,1000G25.15)
end program doublewell_elevels
!=========================================================
subroutine calculate_evs(H,H0,X4,E,WORK,lambda,DIM)
!=========================================================
 implicit none
 integer,parameter          :: P=1000
 integer,parameter          :: LWORK=3*P-1
 real(8),dimension(P,P)     :: H,H0,X4
 real(8),dimension(P)       :: E
 real(8),dimension(LWORK)   :: WORK
 integer                    :: DIM
 real(8)                    :: lambda
 character(1)               :: JOBZ,UPLO
 integer                    :: LDA,INFO,i,j

 call calculate_H(H,H0,X4,lambda,DIM)
 JOBZ='V';UPLO='U'
 call DSYEV(JOBZ,UPLO,DIM,H,P,E,WORK,LWORK,INFO)
 print *,'# ********************** EVEC ******************'
 do j=1,DIM
  write(6,101)'# EVEC ',DIM,lambda,(H(i,j), i=1,DIM)
 enddo
 print *,'# ********************** EVEC ******************'
101 FORMAT(A7,I5,F8.4,1000G14.6)

 if(INFO .ne. 0)then
  print *,'dsyev failed. INFO= ',INFO
  stop
 endif
```

```
end subroutine calculate_evs
!=========================================================
subroutine calculate_H(H,H0,X4,lambda,DIM)
!=========================================================
 implicit none
 integer,parameter        :: P=1000
 real(8),dimension(P,P)   :: H,H0,X4
 integer                  :: DIM
 real(8)                  :: lambda
 integer                  :: i,j

 do j=1,DIM
  do i=1,DIM
   H(i,j)=H0(i,j)+0.25D0*lambda*X4(i,j)
  enddo
 enddo

 print *,'# ********************** H ******************'
 do j=1,DIM
  write(6,102)'# HH ',(H(i,j), i=1,DIM)
 enddo
 print *,'# ********************** H ******************'

102 FORMAT(A5,1000G14.6)
end subroutine calculate_H
!=========================================================
subroutine  calculate_operators(X,X2,X4,iP,P2,H0,DIM)
!=========================================================
 implicit none
 integer,parameter        :: P=1000
 real(8),dimension(P,P)   :: X,X4,X2,iP,P2,H0
 integer                  :: DIM
 integer                  :: i,j,m,n
 real(8),parameter        :: isqrt2=1.0D0/sqrt(2.0D0)

 X =0.0D0;X2=0.0D0;X4=0.0D0
 iP=0.0D0;P2=0.0D0

 do i=1,DIM
  n=i-1 !indices 0,...,DIM-1
! The delta_{n,m+1} term, i.e. m=n-1
  m=n-1 !energy level: n -> i=n+1, m-> j=m+1
  j=m+1
  if(j.ge.1) X (i,j) =  isqrt2*sqrt(DBLE(m+1))
  if(j.ge.1) iP(i,j) =  isqrt2*sqrt(DBLE(m+1))
! The delta_{n,m-1} term, i.e. m=n+1
  m=n+1
  j=m+1
  if(j.le.DIM) X (i,j) =  isqrt2*sqrt(DBLE(m))
```

```
  if(j.le.DIM) iP(i,j) = -isqrt2*sqrt(DBLE(m))
 enddo !do i=1,DIM

 X2 =  MATMUL( X, X)
 P2 = -MATMUL(iP,iP)
 X4 =  MATMUL(X2,X2)

!The Hamiltionian:
 H0 =  0.5D0*(P2-X2)

end subroutine calculate_operators
```

Where is the particle's favorite place when it is in the states $|+\rangle$ and $|-\rangle$? The answer to this question is obtained from the study of the expectation value of the position operator $\langle x\rangle$ in each one of them. We know that when the particle is in one of the energy eigenstates, then we have that

$$\langle x\rangle_n(\lambda) = {}_\lambda\langle n|\, x\, |n\rangle_\lambda = 0 \tag{9.38}$$

because the potential $V(x) = V(-x)$ is even. Therefore

$$\begin{aligned}\langle x\rangle_\pm(\lambda) &= \langle\pm|\, x\, |\pm\rangle \\ &= \frac{1}{\sqrt{2}}\left({}_\lambda\langle 0|\, x\, |0\rangle_\lambda \pm {}_\lambda\langle 1|\, x\, |0\rangle_\lambda \pm {}_\lambda\langle 0|\, x\, |1\rangle_\lambda + {}_\lambda\langle 1|\, x\, |0\rangle_\lambda\right) \\ &= \pm\sqrt{2}\langle 1|\, x\, |0\rangle_\lambda\,, \end{aligned} \tag{9.39}$$

where in the last line we used the relation (9.38) ${}_\lambda\langle 0|\, x\, |0\rangle_\lambda = {}_\lambda\langle 1|\, x\, |1\rangle_\lambda = 0$ and that the amplitudes ${}_\lambda\langle 1|\, x\, |0\rangle_\lambda = {}_\lambda\langle 0|\, x\, |1\rangle_\lambda$. Also[9] we have that ${}_\lambda\langle 1|\, x\, |0\rangle_\lambda > 0$. Therefore, if we have that $|0\rangle_\lambda = \sum_{m=0}^\infty c_m^{(0)}\, |m\rangle$ and $|1\rangle_\lambda = \sum_{m=0}^\infty c_m^{(1)}\, |m\rangle$, we obtain

$$\langle x\rangle_\pm(\lambda) = \pm\sqrt{2}\sum_{m,m'=0}^\infty c_m^{(0)} c_{m'}^{(1)} X_{mm'}\,. \tag{9.40}$$

Given that for finite $N$, the subroutine `DSYEV` returns approximations to the coefficients $c_m^{(n)}$ in the columns of the matrix `H(DIM,DIM)` so that $c_m^{(n)} \approx$ `H(m+1,n+1)`, you may compare the value of $\langle x\rangle_\pm(\lambda)$ with the classical values $x_0 = \pm 1/\sqrt{\lambda}$ as $\lambda$ is increased.

[9] You may convince yourselves by looking at the wave functions in figures 10.4 of chapter 10 and by computing the relevant integrals.

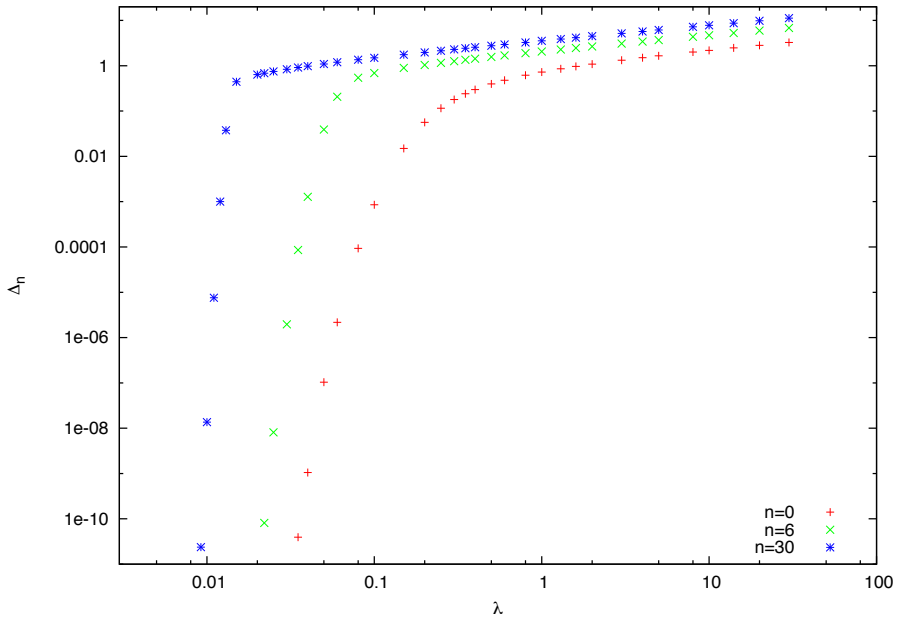



Figure 9.8: Calculation of the difference of the energy levels $\Delta_n = E_{n+1} - E_n$ for $n = 0, 6, 30$ for the double well potential from the program `doublewell.f90`. The difference vanishes as the well becomes deeper with decreasing $\lambda$. The states $|\pm\rangle = (\,|n+1\rangle_\lambda \pm |n\rangle_\lambda)/\sqrt{2}$ are more and more localized to the right or left well respectively.

## 9.5 Problems

9.1 Calculate the matrix $H(\lambda)$ for $N = 2, 3$ analytically. Calculate its eigenvalues for $N = 2$. Compare your results with the numerical values that you obtain from your program.

9.2 Add the necessary code to the program in the file `test.f90` so that it checks that the eigenvectors satisfy their defining relations $\mathbf{A}\,\mathbf{v}_i = \lambda_i \mathbf{v}_i$ and that they form an orthonormal basis $\mathbf{v}_i \cdot \mathbf{v}_j = \delta_{ij}$.

9.3 Calculate $E_5(\lambda)$ and $E_9(\lambda)$ for $\lambda = 0.8, 1.2$ with an accuracy better than $0.01\%$.

9.4 For how large $n$ can you calculate $E_n(\lambda)$ for $\lambda = 1$ with an accuracy better than $2\%$ when $N = 64$?

9.5 Calculate $E_3(\lambda)$ and $E_{12}(\lambda)$ for $0 \leq \lambda \leq 4$ with step $\delta\lambda = 0.2$ by achieving accuracy better than $0.01\%$. How large should $N$ be taken in each case?

9.6 Calculate the expression that gives the matrix elements of the operator $x^4$ in the $|n\rangle$ representation analytically. Modify the program in `anharmonic.f90` in order to incorporate your calculation. Verify that the results are the same and test if it has an effect in the total computation time with and without calculating the eigenvalues and eigenvectors of the Hamiltonian. Compute in each case the dependence of the cpu time on $N$ by computing the exponent (cpu time)$\sim N^a$ for $N = 40 - 1000$.

9.7 Modify the code in the file `anharmonic.f90` so that the arrays `H, X, X4, E, WORK` are `ALLOCATABLE` and their dimension is determined by the variable `DIM` read by the program interactively.
(Hint: Look at the file `anharmonicSPEED.f90`.)

9.8 Make an attempt to reproduce the results of Hioe and Montroll [39] given in table 9.2 for $n = 3$ and $n = 5$. What is the largest value of $\lambda$ that you can study given your computational resources?

9.9 Make an attempt to reproduce the results of Hioe and Montroll [39] given by equation (9.31). Calculate the ground state energy $E_0$ for $200 < \lambda < 20000$ and then fit your results to a function of the form $\lambda^{1/3}(a + b\lambda^{-2/3} + c\lambda^{-4/3})$. What is the accuracy in the calculation of the coefficients $a$, $b$ and $c$ and how good is the agreement with equation (9.31)?

9.10 Modify the code in the file `anharmonic.f90` so that it calculates the expectation values $\langle x^2\rangle_n(N,\lambda)$, $\langle p^2\rangle_n(N,\lambda)$ and the corresponding products $\Delta x \cdot \Delta p$.
(Hint: See the file `anharmonicOBS.f90`.)

9.11 Reproduce the results shown in figure 9.4. Repeat your calculation for $\lambda = 2.0, 10.0, 100.0$. Repeat your calculations for $n = 20$.

9.12 Reproduce the results shown in figure 9.5. Repeat your calculations for $n = 20$.

9.13 Reproduce the results shown in figure 9.6. Repeat your calculation for $n = 3, 7, 12, 18, 24$.

9.14 Write a program that calculates the energy levels of the anharmonic oscillator

$$H(\lambda,\mu) = \frac{1}{2}p^2 + \frac{1}{2}x^2 + \lambda x^4 + \mu x^6 \,. \tag{9.41}$$

Calculate $E_n(\lambda)$ for $n = 0, 3, 8, 20$, $\lambda = 0.2$ and $\mu = 0.2, 0.5, 1.0, 2.0, 10.0$.

9.15 Modify the program of the previous problem so that it calculates the expectation values $\langle x^2\rangle_n(N,\lambda)$, $\langle p^2\rangle_n(N,\lambda)$ and the products $\Delta x \cdot \Delta p$. Calculate the expectation values $\langle x^2\rangle_n(\lambda)$, $\langle p^2\rangle_n(\lambda)$ and $\Delta x \cdot \Delta p$ for $n = 0, 3, 8, 20$, $\lambda = 0.2$ and $\mu = 0.2, 0.5, 1.0, 2.0, 10.0$.

9.16 Use the program `doublewell.f90` in order to calculate the energy level pairs $E_n, E_{n+1}$ for $n = 0, 4, 20$ and $\lambda = 0.2, 0.1, 0.05, 0.02$. Calculate the difference $\Delta_n = E_{n+1} - E_n$ and comment on your results.

9.17 Define the energy values

$$\epsilon_n = -\frac{1}{4\lambda} + \left(n + \frac{1}{2}\right) \,.$$

Compare the results for $E_n, E_{n+1}$ of the previous problem with $\epsilon_n - \Delta_n/2$ and $\epsilon_n + \Delta_n/2$ respectively. Explain your results.

9.18 Modify the program `doublewell.f90`, so that it calculates the expectation values $\langle x\rangle_\pm(\lambda)$ given by equation (9.40). Compare $\langle x\rangle_\pm(\lambda)$ with the classical values $x_0 = \pm 1/\sqrt{\lambda}$ for $\lambda = 0.2, 0.1, 0.05, 0.02, 0.01$.

9.19 Repeat the previous problem when the states $|\pm\rangle = (1/\sqrt{2})(\,|n\rangle_\lambda \pm |n+1\rangle_\lambda)$ for $n = 6$ and $n = 30$.

9.20 For the simple harmonic oscillator, the energy levels are equidistant, i.e. $\Delta_n = E_{n+1} - E_n = 1$, $(\Delta_{n+2} - \Delta_n)/\Delta_n = 0$. Calculate these quantities for the anharmonic oscillator and the double well potential for $\lambda = 1, 10, 100, 1000$ and $n = 0, 8, 20$. What do you conclude from your results?

# Chapter 10

# Time Independent Schrödinger Equation

In this chapter, we will study the time independent Schrödinger equation for a non relativistic particle of mass $m$, without spin, moving in one dimension, in a static potential $V(x)$. We will only study bound states. The solutions in this case yield the discrete energy spectrum $\{E_n\}$ as well as the corresponding eigenstates of the Hamiltonian $\{\psi_n(x)\}$ in position representation.

From a numerical analysis point of view, the problem consists of solving for the eigensystem of a differential equation with boundary conditions. Part of the solution is the energy eigenvalue which also needs to be determined.

As an exercise, we will use two different methods, one that can be applied to a particle in an infinite well with $V(x) = V(-x)$, and one that can be applied to more general cases. The first method is introduced only for educational purposes and the reader may skip section 10.2 to go directly to section 10.3.

## 10.1 Introduction

The wave functions $\psi(x)$, which are the position representation of the energy eigenstates, satisfy the Schrödinger equation

$$-\frac{\hbar^2}{2m}\frac{\partial^2\psi(x)}{\partial x^2} + V(x)\psi(x) = E\psi(x)\,, \tag{10.1}$$

with the normalization condition

$$\langle\psi|\psi\rangle = \int_{-\infty}^{+\infty} \psi^*(x)\psi(x)\,dx = 1\,. \tag{10.2}$$

The Hamiltonian operator is given in position representation by

$$\hat{H} = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(\hat{x})\,, \tag{10.3}$$

and it is Hermitian, i.e. $\hat{H}^\dagger = \hat{H}$. Equation (10.1) is an eigenvalue problem

$$\hat{H}\psi(x) = E\psi(x)\,, \tag{10.4}$$

which, for bound states, has as solutions a discrete set of *real* functions $\psi_n^*(x) = \psi_n(x)$ such that $\hat{H}\psi_n(x) = E_n\psi_n(x)$. The numbers $E_0 \leq E_1 \leq E_2 \leq \ldots$ are real and they are the (bound) energy spectrum of the particle in the potential[4] $V(x)$. The minimum energy $E_0$ is called the ground state energy and the corresponding ground state is given by a non trivial function $\psi_0(x)$. According to the Heisenberg uncertainty principle, in this state the uncertainties in momentum $\Delta p > 0$ and position $\Delta x > 0$ so that $\Delta p \cdot \Delta x \geq \hbar/2$.

The eigenstates $\psi_n(x)$ form an orthonormal basis

$$\langle\psi_n|\psi_m\rangle = \int_{\infty}^{+\infty} \psi_n^*(x)\psi_m(x)\,dx = \delta_{n,m}\,. \tag{10.5}$$

so that any (square integrable) wave function $\phi(x)$ which represents the state $|\phi\rangle$ is given by the linear combination

$$\phi(x) = \sum_{n=0}^{\infty} c_n\psi_n(x)\,. \tag{10.6}$$

The amplitudes $c_n = \langle\psi_n|\phi\rangle = \int_{-\infty}^{+\infty} \psi_n^*(x)\phi(x)\,dx$ are complex numbers that give the probability $p_n = |c_n|^2$ to measure energy $E_n$ in the state $|\phi\rangle$.

For any state $|\phi\rangle$ the function

$$p_\phi(x) = |\phi(x)|^2 = \phi^*(x)\phi(x) \tag{10.7}$$

is the probability density of finding the particle at position $x$, i.e. the probability of detecting the particle in the interval $[x_1, x_2]$ is given by

$$\mathcal{P}_\phi(x_1 < x < x_2) = \int_{x_1}^{x_2} p_\phi(x)\,dx = \int_{x_1}^{x_2} \phi^*(x)\phi(x)\,dx\,. \tag{10.8}$$

[4]The fact that the energy spectrum of the particle is bounded from below depends on the form of the potential. We assume that $V(x)$ is such that $E_0$ is finite. Also, in one dimension, the energy spectrum of a particle for reasonable potentials is non degenerate (see, however, S. Kar, R. Parwani, arXiv:0706.1135.)

The normalization condition (10.2) reflects the conservation of probability (independent of time, respected by the time dependent Schrödinger equation) and the completeness (in this case the certainty that the particle will be observed somewhere on the $x$ axis).

The classical observables $\mathcal{A}(x,p)$ of this quantum mechanical system are functions of the position and the momentum and their quantum mechanical versions are given by operators $\hat{\mathcal{A}}(\hat{x},\hat{p})$. Their expectation values when the system in a state $|\phi\rangle$ are given by

$$\langle\hat{\mathcal{A}}\rangle_\phi = \langle\phi|\,\hat{\mathcal{A}}\,|\phi\rangle = \int_{-\infty}^{+\infty} \phi^*(x)\hat{\mathcal{A}}(\hat{x},\hat{p})\phi(x)\,dx\,. \tag{10.9}$$

From a numerical point of view, the eigenvalue problem (10.1) requires the solution of an ordinary second order differential equation. There are certain differences in this problem compared to the ones studied in previous sections:

- Instead of an initial value problem (i.e. the values of the function and its derivative are given at one point), we have a boundary value problem (values of the function or its derivative given at two different points).
- The eigenvalue (energy) is unknown and should be determined as part of the solution.

As an introduction to such classes of problems, we will present some simple methods which are special to one dimension.

For the numerical solution of the above equation we renormalize $x$, the function $\psi(x)$ and the parameters so that we deal only with dimensionless quantities. Equation (10.1) is rewritten as:

$$\frac{d^2}{dx^2}\psi(x) + \frac{2m}{\hbar^2}(E - V(x))\psi(x) = 0\,. \tag{10.10}$$

Then we choose a length scale $L$ which is defined by the parameters of the problem[2] and we redefine $\tilde{x} = x/L$. We define $\tilde{\psi}(\tilde{x}) = \psi(x)$ $\tilde{\psi}'(\tilde{x}) = d\psi(x)/d\tilde{x} = L\,d\psi(x)/dx$ and we obtain

$$\tilde{\psi}''(\tilde{x}) + \frac{2mL^2}{\hbar^2}(E - V(\tilde{x}L))\tilde{\psi}(\tilde{x}) = 0\,. \tag{10.11}$$

[2]There are $m$, $\hbar$ and the coupling constants in the function $V(x)$. The range of the potential will determine $L$ in some problems and it is given explicitly in potential wells. In potentials of real physical systems, however, this is also determined by the coupling constants.

We define $v(\tilde{x}) = 2mL^2V(x)/\hbar^2 = 2mL^2V(\tilde{x}L)/\hbar^2$, $\epsilon = 2mL^2E/\hbar^2$ and change notation to $\tilde{x} \to x$, $\tilde{\psi} \to \psi$. We obtain

$$\psi''(x) = -(\epsilon - v(x))\psi(x)\,. \tag{10.12}$$

The solutions of equation (10.1) can be obtained from those of equation (10.12) by using the following "dictionary"[3]:

$$x \to \frac{x}{L}\,, \quad E = \frac{\hbar^2}{2mL^2}\epsilon\,, \quad V(x) = \frac{\hbar^2}{2mL^2}v(x/L)\,. \tag{10.13}$$

The dimensionless momentum is defined as $\tilde{p} = -i\partial/\partial\tilde{x} = -iL\partial/\partial x$ and we obtain

$$\tilde{p} = \frac{L}{\hbar}p\,. \tag{10.14}$$

The commutation relation $[x, p] = i\hbar$ becomes $[\tilde{x}, \tilde{p}] = i$. The kinetic energy $T = \dfrac{p^2}{2m}$ is given by

$$T = \frac{\hbar^2}{2mL^2}\tilde{p}^2 = -\frac{\hbar^2}{2mL^2}\frac{\partial^2}{\partial\tilde{x}^2}\,, \tag{10.15}$$

and the Hamiltonian $H = T + V$

$$H = \frac{\hbar^2}{2mL^2}\left(\tilde{p}^2 + v(\tilde{x})\right) = \frac{\hbar^2}{2mL^2}\left(-\frac{\partial^2}{\partial\tilde{x}^2} + v(\tilde{x})\right)\,. \tag{10.16}$$

In what follows, we will omit the tilde above the symbols and write $x$ instead of $\tilde{x}$.

## 10.2 The Infinite Potential Well

The simplest model for studying the qualitative features of bound states is the infinite potential well of width $L$ where a particle is confined within the interval $[-L/2, L/2]$:

$$v(x) = \begin{cases} 0 & |x| < 1 \\ +\infty & |x| \geq 1 \end{cases} \tag{10.17}$$

The length scale chosen here is $L/2$ and the dimensionless variable $x$ corresponds to $x/(L/2)$ when $x$ is measured in length units.

[3]If we normalize the solutions $\tilde{\psi}(\tilde{x})$ of equation (10.12) according to the relation $\int_{-\infty}^{+\infty}\tilde{\psi}^*(\tilde{x})\tilde{\psi}(\tilde{x})d\tilde{x} = 1$, we should also take $\psi(x) = (1/\sqrt{L})\tilde{\psi}(x/L)$ in order to be properly normalized $\int_{-\infty}^{+\infty}\psi^*(x)\psi(x)dx = 1$.

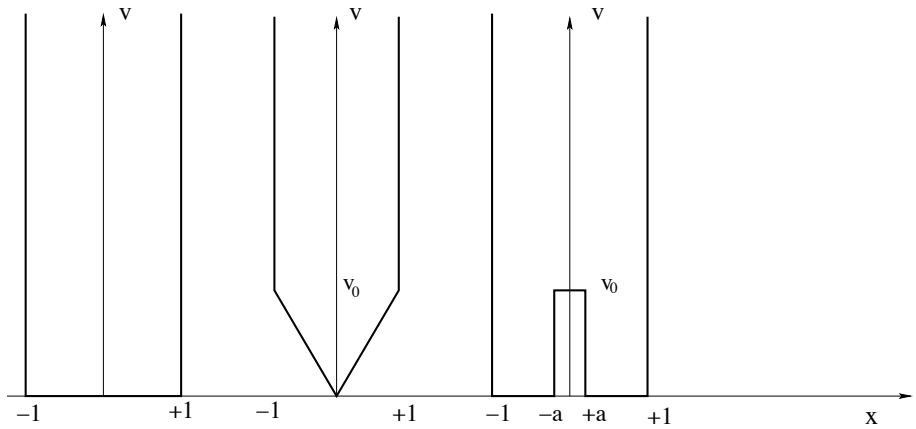



Figure 10.1: The potentials given by equations (10.17), (10.26) and (10.27).

The solution of (10.12) can be easily computed. Due to the symmetry

$$v(-x) = v(x)\,, \tag{10.18}$$

of the potential, the solutions have well defined *parity*. This property will be crucial to the method used below. The method discussed in the next section can also be used on non even potentials.

The solutions are divided into two categories, one with even parity $\psi_n(x) \equiv \psi_n^{(+)}(-x) = \psi_n^{(+)}(x)$ for $n = 1, 3, 5, 7, \ldots$ and one with odd parity $\psi_n(x) \equiv -\psi_n^{(-)}(-x) = \psi_n^{(-)}(x)$ for $n = 2, 4, 6, 8, \ldots$.

$$\psi_n(x) = \begin{cases} \psi_n^{(+)}(x) = \cos\left(\frac{n\pi}{2}x\right) & |x| < 1 \quad n = 1, 3, 5, 7, \ldots \\ \psi_n^{(-)}(x) = \sin\left(\frac{n\pi}{2}x\right) & |x| < 1 \quad n = 2, 4, 6, 8, \ldots \end{cases} \tag{10.19}$$

where

$$\epsilon_n = \left(\frac{n\pi}{2}\right)^2\,, \tag{10.20}$$

and the normalization has been chosen so that[4] $\int_{-1}^{1} (\psi_n(x))^2\, dx = 1$.

The solutions can be found by using the parity of the wave functions. We note that for the positive parity solutions

$$\psi_n^{(+)}(0) = A \qquad \psi_n^{(+)\prime}(0) = 0\,, \tag{10.21}$$

[4]According to the dictionary mentioned in the previous section, for a potential well where $x \in [-L/2, L/2]$ the dimensionless position variable has been chosen to be $x/(L/2) \in [-1, 1]$. Then $E_n = \frac{\hbar^2}{2m(L/2)^2}\epsilon_n = \frac{\hbar^2\pi^2}{2mL^2}n^2$ and $\psi_n^{(+)}(x) = \sqrt{2/L}\cos(n\pi x/L)$, $\psi_n^{(-)}(x) = \sqrt{2/L}\sin(n\pi x/L)$. Note that $\epsilon_n = p_n^2$ according to equations (10.13) and (10.14).

whereas for the negative parity solutions

$$\psi_n^{(-)}(0) = 0 \qquad \psi_n^{(-)\prime}(0) = A\,. \tag{10.22}$$

The constant $A$ depends on the normalization of the wave function. Therefore we can set $A = 1$ originally and then renormalize the wave function so that equation (10.2) is satisfied. If the energy is known, the relations (10.21) and (10.22) can be taken as initial conditions in relation (10.12). By using a Runge–Kutta algorithm we can evolve the solution towards $x = \pm 1$. The problem is that the energy $\epsilon$ is unknown. If the energy is not allowed by the quantum theory we will find that the boundary conditions

$$\psi_n^{(\pm)}(\pm 1) = 0 \tag{10.23}$$

are violated. As we approach the correct value of the energy, we obtain $\psi_n^{(\pm)}(\pm 1) \to 0$.

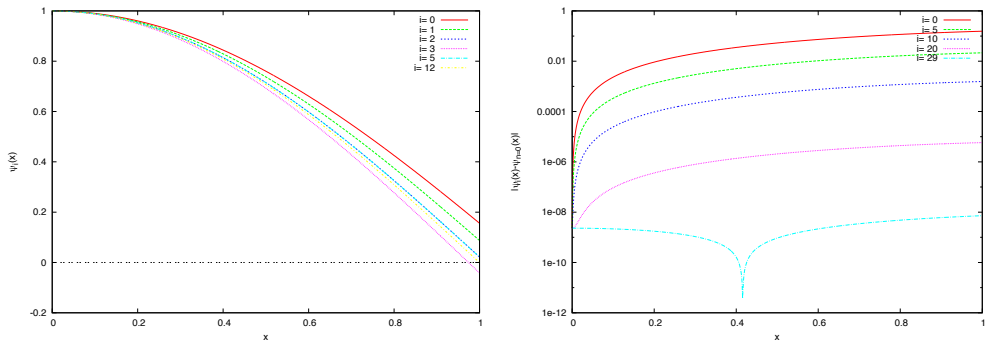



Figure 10.2: Convergence of the solution $\psi_i(x)$ of (10.12) with the potential (10.17) as a function of the number of iterations $i$ in the program `well.f90`. Initially `energy = 2.0` and `parity = 1`. After 29 iterations the solution converges to the ground state $\psi_1(x) = \cos(\pi x/2)$ with energy $\epsilon = (\pi/2)^2$ and with relative accuracy $\sim 10^{-9}$. The bottom plot shows the error as a function of the number of iterations in a logarithmic scale. For $i \equiv$`iter = 1,2,3,5,10,12,20` we obtain `energy = 2.4, 2.6, 2.4, 2.4625, 2.46875, 2.4673828125.`

Therefore we follow the steps described below:

- We choose an initial value for the energy $\epsilon$ that is lower than the one we are looking for. We can use estimates from known solutions of similar looking potential wells or simply start from a value slightly higher than the absolute minimum of the potential.
- We choose the parity of the solution and we set initial conditions according to equations (10.21) and (10.22).

- We evolve the solutions using a 4th order Runge-Kutta method from [5] $x = 0$ to $x = +1$.
- If equation (10.23) is not satisfied, we increase the energy by $\delta\epsilon$ and we repeat.
- We repeat until $\psi_n^{(\pm)}(1)$ changes sign. Then we lower the energy by $\delta\epsilon = -\delta\epsilon/2$.
- The process is ended when $|\psi_n^{(\pm)}(1)| < \delta$ for appropriately chosen small $\delta$.

For the evolution of the solution from $x = 0$ to $x = 1$ we use the 4th order Runge-Kutta method programmed in the file `rk.f90` of chapter 4. We copy the subroutine `RKSTEP` in a local file `rk.f90`. The integration of (10.12) can by done by using the function $\phi(x) \equiv \psi'(x)$

$$\begin{aligned} \psi'(x) &= \phi(x) \\ \phi'(x) &= (v(x) - \epsilon)\psi(x)\,, \end{aligned} \tag{10.24}$$

with the initial conditions

$$\begin{aligned} \psi(0) = 1 \quad &, \quad \phi(0) \equiv \psi'(0) = 0 \quad \text{even parity} \\ \psi(0) = 0 \quad &, \quad \phi(0) \equiv \psi'(0) = 1 \quad \text{odd parity}\,. \end{aligned} \tag{10.25}$$

We use the notation $\psi(x) \to$ `psi`, $\phi(x) \to$ `psip`. The functions `f1` and `f2` correspond to the right hand side of (10.24). They are the derivatives of $\psi(x)$ and $\phi(x)$ respectively and `f1=psip`, `f2=(V-energy)*psi`. The code of `f1` and `f2` is put in a different file so that we can easily reuse the code for many different potentials $v(x)$. The file `wellInfSq.f90` contains the necessary program for the potential of equation (10.17):

```
!=============================================================
!file: wellInfSq.f
!
!Functions used in RKSTEP routine. Here:
!f1 = psip(x) = psi(x)'
!f2 = psip(x)'= psi(x)''
!
!All one has to set is V, the potential
!=============================================================
!———— trivial function: derivative of psi
```

[5] The function in $[-1, 0)$ is determined by the parity of the solution.

```
real(8) function f1(x,psi,psip)
 real(8) :: x,psi,psip
 f1=psip
end function f1
!==============================================================
!-------- the second derivative of wavefunction:
!psip(x)' = psi(x)'' = -(E-V) psi(x)
real(8) function f2(x,psi,psip)
 implicit none
 real(8) :: x,psi,psip,energy,V
 common /params/energy
!-------- potential, set here:
 V  = 0.0D0
!-------- Schroedinger eq: RHS
 f2 = (V-energy)*psi
end function f2
!==============================================================
```

We stress that the energy $\epsilon =$ `energy` is put in a `common block` so that it can be accessed by the main program.

The main program is in the file `well.f90`. The user enters the parameters (`energy, parity, Nx`) and the loop

```
 do while (iter .lt. 10000)
  ............
  if(DABS(psinew)  .le. epsilon) EXIT
  if(psinew*psiold .lt. 0.0D0  ) de = -0.5D0*de
  energy = energy + de
  ............
 enddo                        ! do while
```

exits when $\psi(1) =$`psinew` has an absolute value which is less than `epsilon`, i.e. when the condition (10.23) is satisfied to the desired accuracy. The value of the energy increases up to the point where the sign of the wave function at $x = 1$ changes (`psinew*psiold`$< 0$). Then the value of the energy is overestimated and we change the sign of the step `de` and reduce its magnitude by a half. The algorithm described on page 406 is implemented inside the loop. After exiting the loop, the energy has been determined with the desired accuracy and the rest of the program stores the solution in the array `psifinal(STEPS)`. The results are written to the file `psi.dat`. Note how the variable `parity` is used so that both cases `parity`$= \pm 1$ can be studied. The full program is listed below:

```
!==============================================================
!file: well.f
```

```
!
!Computation of energy eigenvalues and eigenfunctions
!of a particle in an infinite well with V(-x)=V(x)
!
!Input:  energy: initial guess for energy
!        parity: desired parity of solution (+/- 1)
!        Nx-1  : Number of RK4 steps from x=0 to x=1
!Output: energy: energy eigenvalue
!        psi.dat: final psi(x)
!        all.dat: all psi(x) for trial energies
!==========================================================
program even_potential_well
 implicit none
 integer,parameter :: P=10000
 real(8) :: energy,dx,x,epsilon,de
 common /params/energy
 integer :: parity,Nx,iter,i
 real(8) :: psi,psip,psinew,psiold
 real(8) :: psifinal(-P:P),xstep(-P:P)
!------ Input:
 print *,'Enter energy,parity,Nx:'
 read  *, energy,parity,Nx
 if(Nx  .gt. P) stop 'Nx > P'
 if(parity .gt. 0) then
  parity =  1
 else
  parity = -1
 endif
 print *,'# ######################################'
 print *,'# Estart= ',energy,' parity= ',parity
 dx      = 1.0D0/(Nx-1)
 epsilon = 1.0D-6
 print *,'# Nx=  ',Nx ,' dx = ',dx,' eps= ',epsilon
 print *,'# ######################################'

!----- Calculate:
 open(unit=11,file='all.dat')
 iter    = 0
 psiold  = 0.0D0 ! calculated values of psi at x=1
 psinew  = 1.0D0
 de      = 0.1D0*DABS(energy) ! original change in energy
 do while (iter .lt. 10000)
!---------- Initial conditions at x=0
  x      = 0.0D0
  if(parity .eq. 1)then
   psi   = 1.0D0
   psip  = 0.0D0
  else
   psi   = 0.0D0
```

```
   psip  = 1.0D0
  endif
  write(11,*) iter,energy, x, psi,psip
! ---------- Use Runge-Kutta to forward to x=1
  do i=2,Nx
   x     = (i-2)*dx
   call RKSTEP(x,psi,psip,dx)
   write(11,*) iter,energy,x,psi,psip
  enddo                        ! do i=2,Nx
  psinew = psi
  print *,iter, energy, de,psinew
! ---------- Stop if value of psi close to 0
  if(DABS(psinew)  .le. epsilon) EXIT
! ---------- Change direction of energy search:
  if(psinew*psiold .lt. 0.0D0  ) de = -0.5D0*de
  energy = energy + de
  psiold = psinew
  iter   = iter + 1
 enddo                         ! do while
 close(11)
!We found the solution: calculate it once again and store it
 if(parity .eq. 1)then
  psi    = 1.0D0
  psip   = 0.0D0
  node   = 0  ! count number of nodes of function
 else
  psi    = 0.0D0
  psip   = 1.0D0
  node   = 1
 endif
 x               = 0.0D0
 xstep   (0)     = x
 psifinal(0)     = psi ! array that stores psi(x)
 psiold          = 0.0D0
!------- Use Runge-Kutta to move to x=1
 do i=2,Nx
  x              = (i-2)*dx
  call RKSTEP(x,psi,psip,dx)
  xstep   (i-1) = x
  psifinal(i-1) = psi
! ------ Use parity to compute psi(-x)
  xstep   (1-i) = -x
  psifinal(1-i) = parity*psi
!------- Print final solution:
 open(unit=11,file='psi.dat')
 print *,'Final result: E= ',energy,' n= ',node,&
      ' parity= ',parity
 write(11,*)'# E= '          ,energy,' n= ',node,&
      ' parity= ',parity
```

```
 do i=-(Nx-1),(Nx-1)
  write(11,*) xstep(i),psifinal(i)
 enddo
 close(11)
end program even_potential_well
```

The compilation and running of the program can be done with the commands

```
> gfortran  well.f90 wellInfSq.f90 rk.f90 -o well
> ./well
 Enter energy,parity,Nx:
2.0 1 400
 # ########################################
 # Estart=  2.0000000000000000        parity= 1
 # Nx= 400 dx = 2.50626566416E-003 eps= 9.9999999999E-007
 # ########################################
  0   2.0000000000000     0.200000000000      0.15594369476721
  1   2.2000000000000     0.200000000000      8.74448016806986E-2
.............................................
 28   2.4674072265624     1.220703125000E-5 -1.95005436858826E-6
 29   2.4674011230468    -6.103515625000E-6 -7.24621589476086E-9
 Final result: E=    2.4674011230468746    parity=        1
```

The energy is determined to be $\epsilon$ =2.467401123 which can be compared to the exact value $\epsilon = (\pi/2)^2 \approx$ 2.467401100. The fractional error is $\sim 10^{-8}$. The convergence can be studied graphically in figure 10.2.

The calculation of the excited states is done by changing the parity and by choosing the initial energy slightly higher than the one determined in the previous step[6]. The results are in table 10.1. The agreement with the exact result $\epsilon_n = (n\pi/2)^2$ is excellent.

We close this section with two more examples. First, we study a potential well with triangular shape at its bottom

$$v(x) = \begin{cases} v_0|x| & |x| < 1 \\ +\infty & |x| > 1 \end{cases} \tag{10.26}$$

and then a double well potential with

$$v(x) = \begin{cases} v_0 & |x| < a \\ 0 & a < |x| < 1 \\ +\infty & 1 < |x| \end{cases} \tag{10.27}$$

[6]Careful: if the energy levels are too close, we should keep the initial energy constant and change the sign of parity.

| $n$ | $(n\pi/2)^2$ | Square | Triangular | Double Well |
|---|---|---|---|---|
| 1 | 2.467401100 | 2.467401123 | 5.248626709 | 15.294378662 |
| 2 | 9.869604401 | 9.869604492 | 14.760107422 | 15.350024414 |
| 3 | 22.2066099 | 22.2066040 | 27.0690216 | 59.1908203 |
| 4 | 39.47841 | 39.47839 | 44.51092 | 59.96887 |
| 5 | 61.6850275 | 61.6850242 | 66.6384315 | 111.3247375 |
| 6 | 88.82643 | 88.82661 | 93.84588 | 126.37628 |
| 7 | 120.902653 | 120.902664 | 125.878830 | 150.745215 |
| 8 | 157.91367 | 157.91382 | 162.92569 | 194.07578 |
| 9 | 199.859489 | 199.859490 | 204.845026 | 235.017471 |
| 10 | 246.74011 | 246.74060 | 251.74813 | 275.67383 |
| 11 | 298.555533 | 298.555554 | 303.545814 | 331.428306 |
| 12 | 355.3057 | 355.3064 | 360.3107 | 388.7444 |

Table 10.1: Energy eigenvalues for the square, triangular and double well potentials (equations (10.17), (10.26) with $v_0 = 10$ and equation (10.27) with $v_0 = 100$, $a = 0.3$). The agreement of the results for the square potential with the exact ones is excellent. For the other potentials, we note that as we move further from the bottom of the well we obtain energy levels very close to those of the square well: The particle does not feel the influence of the details at the bottom of the well. For the double well potential we obtain $E_1 \approx E_2$ and $E_3 \approx E_4$ according to the analysis on page 413.

where the parameters $v_0, a$ are positive numbers. A qualitative plot of these functions is shown in figure 10.1.

For the triangular potential we take $v_0 = 10$, whereas for the double well potential $v_0 = 100$ and $a = 0.3$. The code in `wellInfSq.f90` is appropriately modified and saved in the files `wellInfTr.f90` and `wellInfDbl.f90` respectively. All we have to do is to change the line computing the value of the potential in the function `f2`. For example the file `wellInfTr.f90` contains the code

```
!------- potential, set here:
 V  = 10.0D0*DABS(x)
```

whereas the file `wellInfDbl.f90` contains the code

```
!------- potential, set here:
 if( DABS(x) .le. 0.3D0)then
  V = 100.0D0
 else
  V = 0.0D0
 endif
```

The analysis is performed in exactly the same way and the results are shown in table 10.1. Note that, for large enough $n$, the energy levels of all the potentials that we studied above tend to have identical values. This happens because, when the particle has energy much larger than $v_0$, the details of the potential at the bottom do not influence its dynamical properties very much. For the triangular potential, the energy levels have higher values than the corresponding ones of the square potential. This happens because, on the average, the potential energy is higher and the potential tends to confine the particle to a smaller region ($\Delta x$ is decreased, therefore $\Delta p$ is increased). This can be seen in figure 10.3 where the wave functions of the particle in each of the two potentials are compared.

Similar observations can be made for the double well potential. Moreover, we note the approximately degenerate energy levels, something which is expected for potentials of this form. This can be understood in terms of the localized states given by the wave functions $\psi_+(x) = (1/\sqrt{2})(\psi_1(x) + \psi_2(x))$ and $\psi_-(x) = (1/\sqrt{2})(\psi_1(x) - \psi_2(x))$. The first one represents a state where the particle is localized in the left well and the second one in the right. This is shown in figure 10.4. As $v_0 \to +\infty$ the two wells decouple and the wave functions $\psi_\pm(x)$ become equal to the energy eigenstate wave functions of two particles in separate infinite square wells of width $1 - a$ with energy eigenvalues $\epsilon_{+,1} = \epsilon_{-,1} = (\pi/(1-a))^2$.

The difference of $\epsilon_1$ and $\epsilon_2$ from these two values is due to the finite $v_0$ (see problem 4).

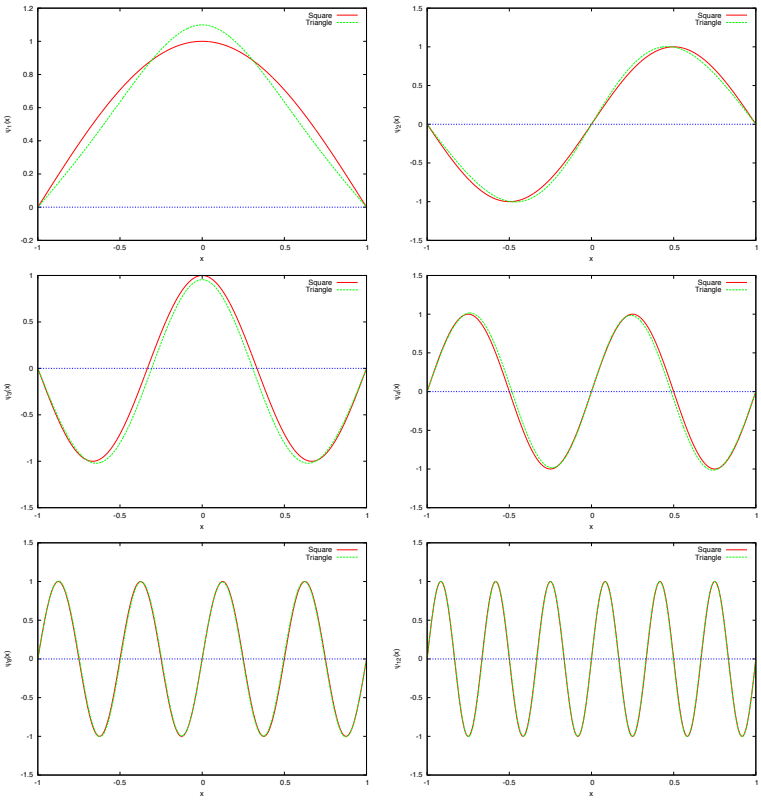



Figure 10.3: The wave functions of the energy eigenstates of the infinite square and triangular well potentials for $n = 1, 2, 3, 4, 8, 12$ given by equations (10.17) and (10.26) with $v_0 = 10$. We observe the influence of the shape of the potential on the wave functions with small $n$, while for $n \geq 8$ the influence becomes weaker.

We will now discuss the limitations of this method. First, the method can be used only on potential wells that are even, i.e. $v(x) = v(-x)$. We used this assumption in equations (10.21) and (10.22) giving the initial conditions for states of well defined parity. When the potential is even, the energy eigenstates have definite parity. The other problem can be understood by solving problem 4: When $v(0) \gg \epsilon$, the wave function is almost zero around $x = 0$ and the integration from $x = 0$ to $x = 1$ will be dominated by numerical errors. The same is true when the particle has to go through high potential barriers.

This method can also we used on potential wells that are not infinite. In that case we can add infinite walls at points that are far enough so

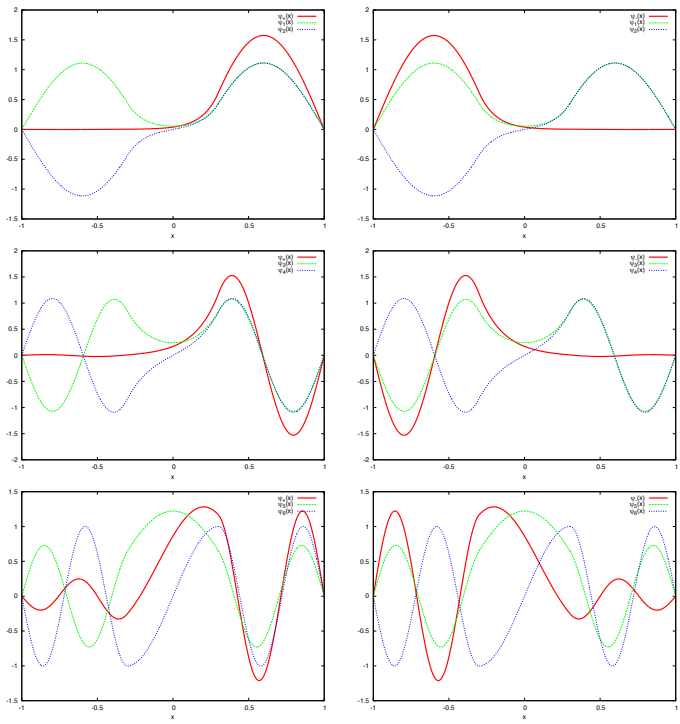


Figure 10.4: The functions $\psi_\pm(x) = (1/\sqrt{2})(\psi_n(x) \pm \psi_{n+1}(x))$ for $n = 1, 3, 5$ for the double well potential (equation (10.27) with $v_0 = 100, a = 0.3$) are plotted using bold red lines. We observe that the more degenerate the states, the stronger the localization of the particle to the left or right well. The other plots are those of the energy eigenfunctions for $n = 1, 2, 3, 4, 5, 6$.

that the wave function is practically zero there. Then the influence of this artificial wall will be negligible (see problem 3).

## 10.3 Bound States

A serious problem with the method discussed in the previous section is that it is numerically unstable. You should have already realized that if you tried to solve problem 3. In that problem, when the walls are moved further than $|x| = 3$, the convergence of the algorithm becomes harder. You can understand this by realizing that in the integration process the solution is evolved from the classically allowed into the classically forbidden region so that an oscillating solution changes into an exponentially damped one. But as $|x| \to +\infty$ there are two solutions, one that is physically acceptable $\psi(x) \sim \mathrm{e}^{-k|x|}$ and one that is diverging $\psi(x) \sim \mathrm{e}^{+k|x|}$

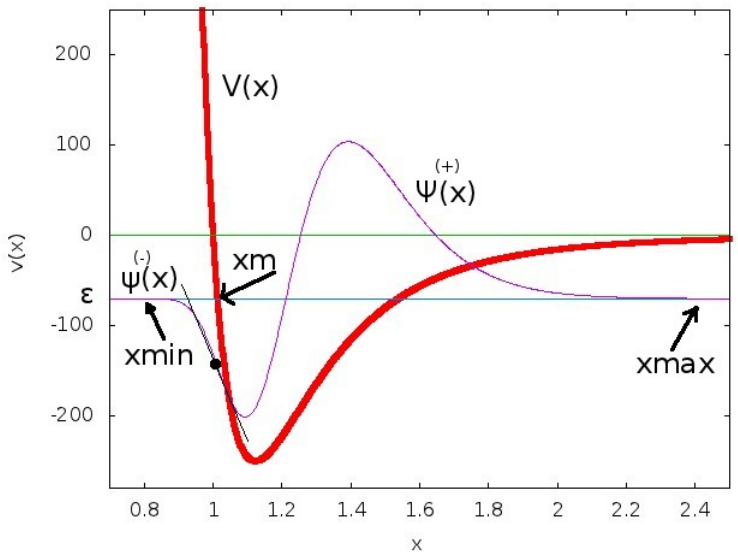



Figure 10.5: Integration of Schrödinger's equation by the use of the algorithm of section 10.3. The wave functions and their derivatives are given small trial values at `xmin` and `xmax` which are in the classically forbidden regions of $x$. The point $x_m$ is calculated from the equation $v(x_m) = \epsilon$. The wave functions are evolved to $x_m$ according to (10.24) and we obtain the solutions $\psi^{(+)}(x)$ and $\psi^{(-)}(x)$. We renormalize $\psi^{(-)}(x)$ so that $\psi^{(+)}(x_m) = \psi^{(-)}(x_m)$ and we vary the energy until the derivatives $\psi^{(+)\prime}(x_m) \approx \psi^{(-)\prime}(x_m)$.

which is not acceptable due to (10.2). Therefore, in order to achieve convergence to the physically acceptable solution, the energy has to be finely tuned, especially when we integrate towards large $|x|$. For this reason it is preferable to integrate from the exponentially damped region towards the oscillating region. The idea is to start integrating from these regions and try to match the solutions and their derivatives at appropriately chosen matching points. The matching is achieved at a point $x_m$ by trying to determine the value of the energy that sets the ratio

$$f(\epsilon) = \frac{\psi^{(+)\prime}(x_m)/\psi^{(+)}(x_m) - \psi^{(-)\prime}(x_m)/\psi^{(-)}(x_m)}{\psi^{(+)\prime}(x_m)/\psi^{(+)}(x_m) + \psi^{(-)\prime}(x_m)/\psi^{(-)}(x_m)} \tag{10.28}$$

equal to zero, within the attainable numerical accuracy. It is desirable to choose a point $x_m$ within the classical region ($\epsilon > v(x)$) and usually we pick a turning point $\epsilon = v(x)$. By renormalizing $\psi^{(\pm)}(x)$ we can always set $\psi^{(+)}(x_m) = \psi^{(-)}(x_m)$, therefore $f(\epsilon) \ll 1$ means that $\psi^{(+)\prime}(x_m) \approx \psi^{(-)\prime}(x_m)$. The denominator of (10.28) sets the scale of the desired accuracy[7] The

[7]If we are unlucky enough to pick a point where $\psi'(x_m) = 0$, this criterion will fail.

idea is depicted in figure 10.5. The algorithm is the following:

- Choose the integration interval `[xmin,xmax]`.
- Choose the initial conditions $\psi^{(-)}(\texttt{xmin})$, $\psi^{(-)\prime}(\texttt{xmin})$, $\psi^{(+)}(\texttt{xmax})$, $\psi^{(+)\prime}(\texttt{xmax})$. This choice depends on the potential $v(x)$. Usually we take `xmin` and `xmax` deep enough in the classically forbidden region and choose the values $\psi^{(-)}(\texttt{xmin})$, $\psi^{(+)}(\texttt{xmax})$ to be zero or exponentially small (e.g. $\sim \mathrm{e}^{-k|x|}$, $k^2 = v(x) - \epsilon$). The corresponding values of the derivatives $\psi^{(-)\prime}(\texttt{xmin})$, $\psi^{(+)\prime}(\texttt{xmax})$ are also taken to be small. The arbitrary normalization of $\psi(x)$ allows these initial values to be chosen in a crude way. The relative sign of the derivatives at large $|x|$ (determined e.g. by the parity of the wave function for even potentials) is also taken care by the renormalization of $\psi^{(-)}(x)$ when applying the matching condition. For an infinite well, the points `xmin,xmax` are the ones where the potential becomes infinite and $\psi^{(-)}(\texttt{xmin}) = \psi^{(+)}(\texttt{xmax}) = 0$.
- Choose the initial value of the energy $\epsilon$ and of the energy variation step $\delta\epsilon$.
- Calculate `xm` from the initial value of the energy and the solution of $v(x) = \epsilon$. Choose the solution that is at the left most side[8].
- Evolve the equations (10.24) from `xmin` to `xm` and obtain the solutions $\psi^{(-)}(x)$,$\psi^{(-)\prime}(x)$.
- Evolve the equations (10.24) from `xmax` to `xm` and obtain the solutions $\psi^{(+)}(x)$,$\psi^{(+)\prime}(x)$.
- Renormalize $\psi^{(-)}(x) \to \psi^{(-)}(x)\left(\psi^{(+)}(\texttt{xm})/\psi^{(-)}(\texttt{xm})\right)$, so that $\psi^{(+)}(\texttt{xm}) = \psi^{(-)}(\texttt{xm})$.
- Compute the ratio $f(\epsilon)$ of equation (10.28).
- If $|f(\epsilon)| < \delta$ for appropriately chosen $\delta > 0$, the calculation ends. The result for the energy eigenvalue and eigenfunction is considered to be determined with adequate accuracy and we may proceed with the analysis of the results.
- If $f(\epsilon)$ changes sign it means that we have crossed the energy eigenvalue. Reverse the direction of search by taking $\delta\epsilon \to -\delta\epsilon/2$.

[8]Note that this point changes when we vary $\epsilon$

- Change the energy $\epsilon \to \epsilon + \delta\epsilon$ and repeat by going back to the fourth step.

When we exit the above loop, the current wave function is a good approximation to the eigenfunction $\psi_n(x)$ corresponding to the eigenvalue $\epsilon_n$. We normalize the wave function according to equation (10.2) and we calculate the expectation values according to (10.9). It is also interesting to determine the number of nodes[9] $n_0$ of the wave function which is related to $n$ by $n = n_0 + 1$.

Our program needs to implement the Runge–Kutta algorithm. We use the routine `RKSTEP` (see page 209) which performs a 4th order Runge–Kutta step. Its code is copied to the file `rk.f90`.

The potential $v(x)$ is coded in the function `V(x)`. The boundary conditions are programmed in the subroutine `boundary(xmin, xmax, psixmin, psipxmin, psixmax, psipxmax)` which returns the values of `psixmin` = $\psi^{(-)}$(`xmin`), `psipxmin` = $\psi^{(-)\prime}$(`xmin`), `psixmax` = $\psi^{(+)}$(`xmax`), `psipxmax` = $\psi^{(-)\prime}$(`xmax`) to the calling program. These routines are put in a separate file for each potential that we want to study. The name of the file is related to the form of the potential, e.g. we choose `schInfSq.f90` for the infinite potential well of (10.17). The same file contains the code for the functions `f1, f2`:

```
!=============================================================
!file: schInfSq.f
!
!Functions used in RKSTEP routine. Here:
!f1 = psip(x) = psi(x)'
!f2 = psip(x)'= psi(x)''
!
!One has to set:
!  1. V(x), the potential
!  2. The boundary conditions for psi,psip at x=xmin and x=xmax
!
!=============================================================
!------ potential:
real(8) function V(x)
 implicit none
 real(8) :: x
 V = 0.0D0
end function V
!------ boundary conditions:
subroutine &
```

[9]The number of points $x$ for which $\psi(x) = 0$ and `xmin` $< x <$ `xmax`. The relation $n = n_0 + 1$ sets $\epsilon_1$ to be the ground state for which $n_0 = 0$.

```
 boundary(xmin,xmax,psixmin,psipxmin,psixmax,psipxmax)
 implicit none
 real(8) :: xmin,xmax,psixmin,psipxmin,psixmax,psipxmax,V
!for infinite square well we set psi=0 at boundary
!and psip=+/-1
 psixmin  =  0.0D0
 psipxmin =  1.0D0
 psixmax  =  0.0D0
 psipxmax = -1.0D0
!------ Initial values at xmin and xmax
end subroutine boundary
!==========================================================
!==========================================================
!------ trivial function: derivative of psi
real(8) function f1(x,psi,psip)
 real(8) :: x,psi,psip
 f1=psip
end function f1
!==========================================================
!------ the second derivative of wavefunction:
!psip(x)' = psi(x)'' = -(E-V) psi(x)
real(8) function f2(x,psi,psip)
 implicit none
 real(8) :: x,psi,psip,energy,V
 common /params/energy
!------ Schroedinger eq: RHS
 f2 = (V(x)-energy)*psi
end function f2
!==========================================================
```

We note that if the potential becomes infinite for $x <$ `xmin` and/or $x >$`xmax`, then this will be determined by the boundary conditions at `xmin` and/or `xmax`.

The main program is in the file `sch.f90`. The code is listed below and it includes the function `integrate(psi, dx, Nx)` used for the normalization of the wave function. It performs a numerical integration of the square of a function whose values `psi(i) i=1,...,Nx` are given at an odd number of `Nx` equally spaced points by a distance `dx` using Simpson's rule.

```
!==========================================================
!
! File: sch.f90
!
! Integrate 1d Schrodinger equation from xmin to xmax.
! Determine energy eigenvalue and eigenfunction by matching
```

```
! evolving solutions from xmin and from xmax at a point xm.
! Matching done by equating values of functions and their
! derivatives at xm. The point xm chosen at the left most
! turning point of the potential at any given value of the
! energy. The potential and boundary conditions chosen in
! different file.
! ------------------------------------------------------------
! Input:  energy: Trial value of energy
!         de: energy step, if matching fails de -> e+de, if
!             logderivative changes sign     de -> -de/2
!         xmin, xmax, Nx
! ------------------------------------------------------------
! Output: Final value of energy, number of nodes of
!    wavefunction in stdout
!    Final eigenfunction in file psi.dat
!    All trial functions and energies in file all.dat
!============================================================
program schroedinger_equation_1D
 implicit none
 integer,parameter :: P=20001
 integer ::  Nx,NxL,NxR
 real(8) ::  psi(P),psip(P)
 real(8) ::  dx
 real(8) ::  xmin,xmax,xm       !left/right/matching points
 real(8) ::  psixmin,psipxmin,psixmax,psipxmax
 real(8) ::  psileft ,psiright ,psistep,psinorm
 real(8) ::  psipleft,psipright,psipstep
 real(8) ::  energy,de,epsilon,integrate
 common/params/energy
 real(8) ::  matchlogd,matchold,psiold,norm,x
 integer ::  iter,i,imatch,nodes
 real(8) ::  V
!---------- Input:
 print *,'# Enter energy,de,xmin,xmax,Nx'
 read  *,energy,de,xmin,xmax,Nx
!--- need even intervals for normalization integration
 if( mod(Nx,2).eq.0)Nx=Nx+1
 if( Nx .gt. P      ) stop 'Fatal Error: Nx>P'
 if( xmin  .ge. xmax) stop 'Error: xmin >= xmax'
 dx      = (xmax - xmin)/(Nx-1)
 epsilon = 1.0D-6
 call boundary(xmin,xmax,psixmin,psipxmin,psixmax,psipxmax)
 print *,'# ######################################'
 print *,'# Estart= ',energy, ' de= ',de
 print *,'# Nx=  ',Nx ,' eps= ',epsilon
 print *,'# xmin= ',xmin,' xmax= ',xmax, ' dx= ',dx
 print *,'# psi(xmin)= ',psixmin,' psip(xmin)= ',psipxmin
 print *,'# psi(xmax)= ',psixmax,' psip(xmax)= ',psipxmax
 print *,'# ######################################'
```

```
!------ Calculate:
 open(unit=11,file='all.dat')
 matchold   = 0.0d0
 do iter=1,10000
!------ Determine matching point at turning point from the left:
  imatch = -1
  do i=1,Nx
   x = xmin + (i-1)*dx
   if( imatch .lt. 0 .and. (energy-V(x)) .gt. 0.0D0) imatch = i
  enddo
  if( imatch .le. 100 .or. imatch .ge. Nx-100) imatch = Nx/5
  xm     = xmin + (imatch-1)*dx
  NxL = imatch
  NxR = Nx-imatch+1
!------ Evolve wavefunction from the left:
  psi  (1)  = psixmin
  psip (1)  = psipxmin
  psistep   = psixmin
  psipstep  = psipxmin
  do i=2,NxL
   x        = xmin + (i-2)*dx !this is x before the step
   call RKSTEP(x,psistep,psipstep, dx)
   psi (i)  = psistep
   psip(i)  = psipstep
  enddo
! use this to normalize eigenfunction to match at xm
  psinorm   = psistep
  psipleft  = psipstep
!------ Evolve wavefunction from the right:
  psi (Nx)  = psixmax
  psip(Nx)  = psipxmax
  psistep   = psixmax
  psipstep  = psipxmax
  do i=2,NxR
   x        = xmax - (i-2)*dx
   call RKSTEP(x,psistep,psipstep,-dx)
   psi (Nx-i+1) = psistep
   psip(Nx-i+1) = psipstep
  enddo
  psinorm       = psistep/psinorm
  psipright     = psipstep
!------ Renormalize psil so that psil(xm)=psir(xm)
  do i=1,NxL-1
   psi (i)      = psinorm * psi (i)
   psip(i)      = psinorm * psip(i)
  enddo
  psipleft      = psinorm * psipleft
!------ print current solution:
  do i=1,Nx
```

```
    x = xmin + (i-1)*dx
    write(11,*)iter,energy,x,psi(i),psip(i)
   enddo
!------ matching using derivatives:
!Careful: this can fail if psi'(xm) = 0 !! (use also |de|<1e-6
!criterion)
   matchlogd = &
        (psipright-psipleft)/(DABS(psipright)+DABS(psipleft))
   print *,'# iter,energy,de,xm,logd: ',&
        iter,energy,de,xm,matchlogd
!------ Exit condition:
   if(DABS(matchlogd).le.epsilon.or.DABS(de/energy).lt.1.0D-12)&
        EXIT
   if( matchlogd * matchold .lt. 0.0D0) de = -0.5D0*de
   energy    = energy + de
   matchold  = matchlogd
  enddo ! do iter=1,10000
  close(11)
!-------------------------------------------------------------
!------ Solution has been found and now it is stored:
  norm     = integrate(psi,dx,Nx)
  norm     = 1.0D0/sqrt(norm)
  do i=1,Nx
   psi(i)  = norm*psi(i)
  enddo
!------ Cound number of zeroes, add one and get energy level:
  nodes    = 1
  psiold   = psi(1)
  do i=2,Nx-1
   !should be 0 within epsilon
   if(  DABS(psi(i)) .gt.epsilon)then
    if( psiold*psi(i).lt.0.0D0  )nodes = nodes+1
    psiold = psi(i)
   endif
  enddo !i=2,Nx-1
!------- Print final solution:
  open(unit=11,file='psi.dat')
  print *,'Final result: E= ',energy,' n= ',nodes,&
       ' norm = '       ,norm
  if( DABS(matchlogd) .gt. epsilon) print *&
       ,'Final result: SOS: logd>epsilon. logd= ',matchlogd
  write(11,*)'# E= '          ,energy,' n= ',nodes,&
       ' norm = '       ,norm
  do i=1,Nx
   x = xmin + (i-1)*dx
   write(11,*) x,psi(i)
  enddo
  close(11)
end program schroedinger_equation_1D
```

```
!==========================================================
!Simpson's rule to integrate psi(x)*psi(x) for proper
!normalization. For n intervals of width dx (n even)
!Simpson's rule is:
!int(f(x)dx) =
! (dx/3)*(f(x_0)+4 f(x_1)+2 f(x_2)+...+4 f(x_{n-1})+f(x_n))
!
!Input:   Discrete values of function psi(Nx)
!         Integration step dx
!Returns: Integral(psi(x)psi(x) dx)
!==========================================================
real(8) function integrate(psi,dx,Nx)
 implicit none
 integer :: Nx
!------------- Note: we need P due to geometry of array
 real(8) :: psi(Nx),dx
!-------------
 real(8) :: int
 integer :: i
!----- zeroth order point:
 i    = 1
 int  = psi(i)*psi(i)
!----- odd  order points (i=k+1 is even):
 do i=2,Nx-1,2
  int = int + 4.0D0*psi(i)*psi(i)
 enddo
!----- even order points:
 do i=3,Nx-2,2
  int = int + 2.0D0*psi(i)*psi(i)
 enddo
!----- last point:
 i    = Nx
 int  = int + psi(i)*psi(i)
!----- measure normalization:
 int  = int*dx/3.0D0
!----- final result:
 integrate = int
end function integrate
!==========================================================
```

The reproduction of the results of the previous section for the infinite potential well is left as an exercise. The compilation and running of the program can be done with the commands:

```
> gfortran sch.f90 schInfSq.f90 rk.f90 -o s
> ./s
 # Enter energy,de,xmin,xmax,Nx
1 0.5 -1 1 2000
```

```
# ######################################
# Estart=     1.000   de=     0.5
# Nx=         2001    eps=    1.0E-006
# xmin=      -1.000   xmax=   1.000  dx=   1.000E-003
# psi(xmin)= 0.000    psip(xmin)=     1.000
# psi(xmax)= 0.000    psip(xmax)=    -1.000
# ######################################
# iter,energy,de,xm,logd:  1 1.0000   0.500     -0.601  -0.9748
# iter,energy,de,xm,logd:  2 1.5000   0.500     -0.601  -0.6412
.....
# iter,energy,de,xm,logd: 30 2.4674  -3.815E-6 -0.601  -1.0E-6
# iter,energy,de,xm,logd: 31 2.4674   1.907E-6 -0.601   2.7E-7
Final result: E= 2.467401504516602 n= 1 norm = 1.5707965025
```

We set `xmin= -1, xmax = 1, Nx= 2000` and $\epsilon = 1$, $\delta\epsilon = 0.5$. The energy of the ground state is found to be $\epsilon_1 = 2.4674015045166016$. The wave function is stored in the file `psi.dat` and can be plotted with the `gnuplot` command

```
gnuplot> plot "psi.dat" using 1:2 with lines
```

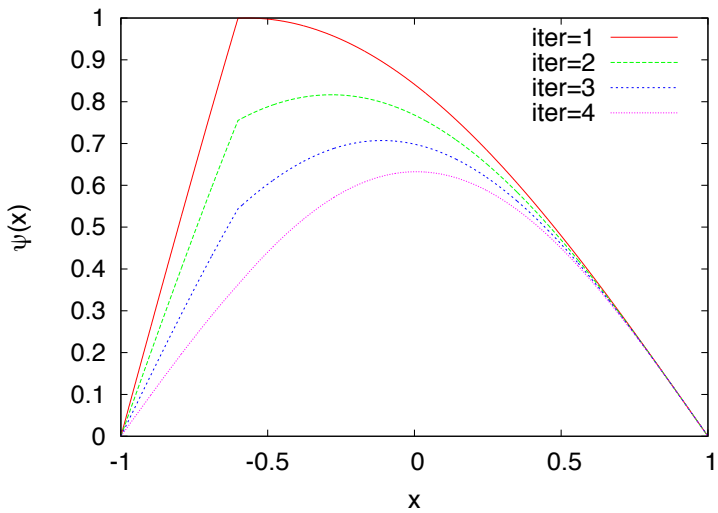



Figure 10.6: The convergence of the solutions to the solution of Schrödinger's equation for the ground state of the infinite potential well according to the discussion on page 424.

The functions computed during the iterations of the algorithm are stored in the file `all.dat`. The first column is the iteration number (here we

have `iter = 0, ... 31`) and we can easily filter each one of them with the commands

```
gnuplot> plot   "<awk '$1==1' all.dat" using 3:4 w l t "iter=1"
gnuplot> replot "<awk '$1==2' all.dat" using 3:4 w l t "iter=2"
gnuplot> replot "<awk '$1==3' all.dat" using 3:4 w l t "iter=3"
gnuplot> replot "<awk '$1==4' all.dat" using 3:4 w l t "iter=4"
.....
```

which reproduce figure 10.6.

## 10.4 Measurements

The action of an operator $\hat{\mathcal{A}}(\hat{x}, \hat{p})$ on a state $|\psi\rangle$ can be easily calculated in the position representation by its action on the corresponding wave function $\psi(x)$. The action of the operators

$$\hat{x}\psi(x) = x\psi(x) \qquad \hat{p}\psi(x) = -i\frac{\partial}{\partial x}\psi(x) \tag{10.29}$$

yield[10]

$$\hat{\mathcal{A}}(\hat{x}, \hat{p})\psi(x) = \mathcal{A}(x, -i\frac{\partial}{\partial x})\psi(x)\,. \tag{10.30}$$

Using equation (10.9) we can calculate the expectation value $\langle\mathcal{A}\rangle$ of the operator $\mathcal{A}$ when the system is at the state $|\psi\rangle$. Interesting examples are the observables "position" $x$, "position squared" $x^2$, "momentum" $p$, "momentum squared" $p^2$, "kinetic energy" $T$, "potential energy" $V$, "energy" or "Hamiltonian" $H = T + V$ whose expectation values are

[10]We do not consider ordering problems of operators formed by products of non commuting operators, e.g. $xp^2$.

given by the relations

$$
\begin{aligned}
\langle x \rangle &= \int_{-\infty}^{+\infty} \psi^*(x)\, x\, \psi(x)\, dx \\
\langle x^2 \rangle &= \int_{-\infty}^{+\infty} \psi^*(x)\, x^2\, \psi(x)\, dx \\
\langle p \rangle &= \int_{-\infty}^{+\infty} \psi^*(x) \left( -i \frac{\partial}{\partial x} \right) \psi(x)\, dx \\
\langle p^2 \rangle &= \int_{-\infty}^{+\infty} \psi^*(x) \left( -\frac{\partial^2}{\partial x^2} \right) \psi(x)\, dx \\
\langle T \rangle &= \frac{\hbar^2}{2mL^2} \int_{-\infty}^{+\infty} \psi^*(x) \left( -\frac{\partial^2}{\partial x^2} \right) \psi(x)\, dx \\
\langle V \rangle &= \frac{\hbar^2}{2mL^2} \int_{-\infty}^{+\infty} \psi^*(x)\, v(x)\, \psi(x)\, dx \\
\langle H \rangle &= \frac{\hbar^2}{2mL^2} \int_{-\infty}^{+\infty} \psi^*(x) \left( -\frac{\partial^2}{\partial x^2} + v(x) \right) \psi(x)\, dx\,. \qquad (10.31)
\end{aligned}
$$

We remind the reader that we used the dimensionless $x, p$ as well as equations (10.15) and (10.16). Especially interesting are the "uncertainties" $\Delta x^2 = \langle x^2 \rangle - \langle x \rangle^2$, $\Delta p^2 = \langle p^2 \rangle - \langle p \rangle^2$ that satisfy the inequality ("Heisenberg's uncertainty relation")

$$\Delta x \cdot \Delta p \geq \frac{1}{2}\,. \qquad (10.32)$$

In the previous section we described how to calculate numerically the eigenfunctions of the Hamiltonian. If $\hat{H}\psi(x) = E\psi(x)$, we obtain that $\langle H \rangle = (1/2mL^2)\epsilon$. Other operators need a numerical approximation for the calculation of their expectation values. If the values of the wave function are given at $N$ equally spaced points $x_1, x_2, \ldots, x_N$, then we obtain

$$\frac{\partial \psi(x_i)}{\partial x} \approx \frac{\psi(x_{i+1}) - \psi(x_{i-1})}{2h} \qquad (10.33)$$

where $h = x_{i+1} - x_i$ and

$$\frac{\partial^2 \psi(x_i)}{\partial x^2} \approx \frac{\psi(x_{i+1}) - 2\psi(x_i) + \psi(x_{i-1})}{h^2}\,. \qquad (10.34)$$

Both equations entail an error of the order of $\mathcal{O}(h^2)$. Special care should be taken at the endpoints of the interval $[x_1, x_N]$. As a first approach we

will use the naive approximations[11]

$$\begin{aligned} \frac{\partial\psi(x_1)}{\partial x} &\approx \frac{\psi(x_2)-\psi(x_1)}{h} \\ \frac{\partial\psi(x_N)}{\partial x} &\approx \frac{\psi(x_N)-\psi(x_{N-1})}{h} \end{aligned} \tag{10.35}$$

and

$$\begin{aligned} \frac{\partial^2\psi(x_1)}{\partial x^2} &\approx \frac{\psi(x_3)-2\psi(x_2)+\psi(x_1)}{h^2} \\ \frac{\partial^2\psi(x_N)}{\partial x^2} &\approx \frac{\psi(x_N)-2\psi(x_{N-1})+\psi(x_{N-2})}{h^2}\,. \end{aligned} \tag{10.36}$$

The relevant program that calculates $\langle x\rangle$, $\langle x^2\rangle$, $\langle p\rangle$, $\langle p^2\rangle$, $\Delta x$, $\Delta p$ can be found in the file `observables.f90` and is listed below:

```
!==========================================================
!
! File observables.f90
! Compile: gfortran observables.f90 -o o
! Usage:    ./o <psi.dat>
!
! Read in a file with a wavefunction in the format of psi.dat:
! # E= <energy> ....
! x1  psi(x1)
! x2  psi(x2)
! ............
!
! Outputs expectation values:
! normalization Energy <x> <p> <x^2> <p^2> Dx Dp DxDp
! where Dx = sqrt(<x^2>-<x>^2) Dp = sqrt(<p^2>-<p>^2)
!  DxDp = Dx * Dp
!
!==========================================================
program observables_expectation
 implicit none
 integer,parameter :: P=50000
 integer Nx,i
 real(8) :: xstep(P),psi(P),obs(P)
 real(8) :: xav, pav, x2av, p2av, Dx, Dp, DxDp,energy,h,norm
 real(8) :: integrate
```

[11]See the files `observables.f90`, `Derivatives.nb` of the accompanying software. There you can find formulas that have errors of $\mathcal{O}(h^2)$. In the examples discussed below, the influence of the $\mathcal{O}(h)$ error on the results is approximately at the fourth significant digit.

```
 character(20) ::  psifile,scratch

!the first argument of the command line must be the path
!to the file with the wavefunction. (GNU fortran extension...)
 if( iargc() .ne. 1) stop 'Usage: o <filename>'
 call getarg(1,psifile)
!If the file does not exist, we go to label 100 (stop):
 open(unit=11,file=psifile,status='OLD',err=100)
 print *,"# reading wavefunction from file:", psifile
!we read the first comment line from the file:
 read(11,*) scratch,scratch,energy
!-------------------------------------------------------------
!Input data: psi(x)
 Nx = 1
 do while(.TRUE.)
  if(Nx .ge. P) stop 'Too many points'
  read(11,*,end=101) xstep(Nx),psi(Nx)
  Nx = Nx+1
 enddo !do while(.TRUE.)
101 continue
 Nx = Nx - 1
 if(mod(Nx,2) .eq. 0) Nx = Nx - 1
 h  = (xstep(Nx)-xstep(1))/(Nx-1)
!-------------------------------------------------------------
!Calculate:
!---------- norm:
 do i=1,Nx
  obs(i) = psi(i)*psi(i)
 enddo
 norm = integrate(obs,h,Nx)
!---------- <x> :
 do i=1,Nx
  obs(i) = xstep(i)*psi(i)*psi(i)
 enddo
 xav = integrate(obs,h,Nx)/norm
!---------- <p>/i :
 obs(1) = psi(1)*(psi(2)-psi(1))/h
 do i=2,Nx-1
  obs(i) = psi(i)*(psi(i+1)-psi(i-1))/(2.0D0*h)
 enddo
 obs(Nx) = psi(Nx)*(psi(Nx)-psi(Nx-1))/h
 pav = -integrate(obs,h,Nx)/norm
!--------- <x^2>
 do i=1,Nx
  obs(i) = xstep(i)*xstep(i)*psi(i)*psi(i)
 enddo
 x2av = integrate(obs,h,Nx)/norm
!-------- <p^2>
 obs(1) = psi(1)*(psi(3)-2.0D0*psi(2)+psi(1))/(h*h)
```

```
 do i=2,Nx-1
  obs(i) = psi(i)*(psi(i+1)-2.0D0*psi(i)+psi(i-1))/(h*h)
 enddo
 obs(Nx) = psi(Nx)*&
      (psi(Nx)-2.0D0*psi(Nx-1)+psi(Nx-2))/(h*h)
 p2av = -integrate(obs,h,Nx)/norm
!-------- Dx
 Dx = sqrt(x2av - xav*xav)
!-------- Dp
 Dp = sqrt(p2av - pav*pav)
!-------- Dx . Dp
 DxDp = Dx*Dp
!print results:
 print *,'# norm E <x> <p>/i <x^2> <p^2> Dx Dp DxDp'
 print '(10G25.17)',norm,energy,xav,pav,x2av,p2av,Dx,Dp,DxDp
 stop !normal execution ends here. Error messages follow
100 stop 'Cannot open file'
end program observables_expectation
!==========================================================
!
!Simpson's rule to integrate psi(x)*psi(x) for proper
!normalization. For n intervals of width dx (n even)
!Simpson's rule is:
!int(f(x)dx) =
! (dx/3)*(f(x_0)+4 f(x_1)+2 f(x_2)+...+4 f(x_{n-1})+f(x_n))
!
!Input:   Discrete values of function psi(Nx)
!         Integration step dx
!Returns: Integral(psi(x)psi(x) dx)
!==========================================================
real(8) function integrate(psi,dx,Nx)
 implicit none
 integer :: Nx
 real(8) :: psi(Nx),dx
 real(8) :: int
 integer i
!----- zeroth order point:
 i    = 1
 int  = psi(i)
!----- odd  order points (i=k+1 is even):
 do i=2,Nx-1,2
  int = int + 4.0D0*psi(i)
 enddo
!----- even order points:
 do i=3,Nx-2,2
  int = int + 2.0D0*psi(i)
 enddo
!----- last point:
 i    = Nx
```

```
 int  = int + psi(i)
!------ measure normalization:
 int  = int*dx/3.0D0
!------ final result:
 integrate = int
end function integrate
!=============================================================
```

The program needs to read in the wave function at the points $x_1, \ldots, x_{\mathtt{Nx}}$ in the format produced by the program in `sch.f90`. The first line should have the energy written at the 3rd column, whereas from the 2nd line and on there should be two columns with the $(x_i, \psi(x_i))$ pairs. It is not necessary to have the wave function properly normalized, the program will take care of it. If this data is stored in a file `psi.dat`, then the program can be used by running the commands

```
> gfortran  observables.f90 -o obs
> ./obs psi.dat
```

The program prints the normalization constant of $\psi(x)$, the value of the energy[12], $\langle x\rangle$, $\langle x^2\rangle$, $\langle p\rangle/i$, $\langle p^2\rangle$, $\Delta x$, $\Delta p$ and $\Delta x \cdot \Delta p$ to the `stdout`.

Some details about the program: In order to read in the data from the file `psi.dat` we use the functions `iargc()`, `getarg(n,string)`. The former returns the number of arguments of the command line and the latter stores the `n`-th argument to the `CHARACTER` variable `string`. Therefore, the statements

```
 character(20) ::  psifile,scratch
 if( iargc() .ne. 1) stop 'Usage: o  <filename>'
 call getarg(1,psifile)
```

stop the program if the command line does not have exactly one argument and store the first argument to the variable `file`.

The command

```
 open(unit=11,file=psifile,status='OLD',err=100)
100 stop 'Cannot open file'
```

opens a file which should already exist (`status='OLD'`), otherwise an error message is issued. The option `err=100` transfers the control of the program to the statement labeled `'100'`. In the example shown above,

[12]The one read from the file. It is not calculated from the data.

the program stops and prints an error message `'Cannot open filename'` to the `stdout`.

The commands

```
 Nx = 1
 do while(.TRUE.)
  read(11,*,end=101) xstep(Nx),psi(Nx)
  Nx = Nx+1
 enddo !do while(.TRUE.)
101 continue
```

read the opened file line by line. The option `end=101` at the statement `read(11,*,end=101)` transfers the control of the program to the labeled statement with label `101` (i.e. outside the `do` loop) when we reach the end of file.

The rest of the commands are applications of equations (10.33), (10.34), (10.35) and (10.36) to the formulas (10.31) and the reader is asked to study them carefully. The program uses the function `integrate` in order to perform the necessary integrals.

## 10.5 The Anharmonic Oscillator - Again...

In the previous chapter 9 we studied the quantum mechanical harmonic and anharmonic oscillator in the representation of the energy eigenstates of the harmonic oscillator $|n\rangle$. In this section we will revisit the problem by using the position representation. We will calculate the eigenfunctions $\psi_{n,\lambda}(x)$ that diagonalize the Hamiltonian (9.15), which are the solutions of the Schrödinger equation. By setting $L = \sqrt{\hbar/m\omega}$ in equation (10.13), equation (10.12) becomes

$$\psi''(x) = -(\epsilon - v(x))\psi(x)\,, \tag{10.37}$$

where $v(x) = x^2 + 2\lambda x^4$. For $\lambda = 0$ we obtain the harmonic oscillator with

$$\psi_n(x) = \frac{1}{\sqrt{2^n n! \sqrt{\pi}}} \mathrm{e}^{-x^2/2} H_n(x)\,, \epsilon_n = 2\left(n + \frac{1}{2}\right)\,, \tag{10.38}$$

where $H_n(x)$ are the Hermite polynomials.

We start with the simple harmonic oscillator where the exact solution is known. The potential and the initial conditions are programmed in the file `schHOC.f90`. The changes that we need to make concern the functions `V(x)`, `boundary(xmin, xmax, psixmin, psipxmin, psixmax, psipxmax)`:

```
!=============================================================
!file: schHOC.f
!..............
!------ potential:
real(8) function V(x)
 implicit none
 real(8) :: x
 V = x*x
end function V
!------ boundary conditions:
subroutine &
 boundary(xmin,xmax,psixmin,psipxmin,psixmax,psipxmax)
 implicit none
 real(8) :: xmin,xmax,psixmin,psipxmin,psixmax,psipxmax,V

 psixmin    =  exp(-0.5D0*xmin*xmin)
 psipxmin   = -xmin*psixmin
 psixmax    =  exp(-0.5D0*xmax*xmax)
 psipxmax   = -xmax*psixmax
end subroutine boundary
!=============================================================
................
```

The code omitted at the dots is identical to the one discussed in the previous section. The initial conditions are inspired by the asymptotic behavior of the solutions to Schrödinger's[13] equation $\psi_0(x) \sim \mathrm{e}^{-x^2/2}$, $\psi_n'(x) \sim -x\psi_n(x)$. You are encouraged to test the influence of other choices on the results. The results are depicted in figure 10.7 where, besides the qualitative agreement, their difference from the known values (10.38) is also shown. This difference turns out to be of the order of $10^{-11}$–$10^{-7}$. The values of the energy $\epsilon_n$ for $n \leq 14$ are in agreement with (10.38) with relative accuracy better than $10^{-9}$.

Then we calculate the expectation values $\langle x\rangle$, $\langle x^2\rangle$, $\langle p\rangle$, $\langle p^2\rangle$, $\Delta x$ and $\Delta p$. These are easily calculated using equations (9.4) and (9.8). We see that $\langle x\rangle = \langle n|\,(a^\dagger + a)/\sqrt{2}\,|n\rangle = 0$, $\langle p\rangle = \langle n|\,i(a^\dagger - a)/\sqrt{2}\,|n\rangle = 0$, whereas

$$\langle x^2\rangle = \langle p^2\rangle = \langle n|\,\frac{1}{2}(a^\dagger a + aa^\dagger)\,|n\rangle = \left(n + \frac{1}{2}\right) . \tag{10.39}$$

The program `observables.f90` calculates $\langle x\rangle = 0$ with accuracy $\sim 10^{-6}$ and $\langle p\rangle = 0$ with accuracy $\sim 10^{-11}$. The expectation values $\langle x^2\rangle$, $\langle p^2\rangle$ are shown in table 10.2.

[13] In fact $\psi_n(x) \sim x^n \mathrm{e}^{-x^2/2}$ which we neglect. This does not influence the results for the values of $n$ studied here. Examine if this is necessary for larger values of $n$.

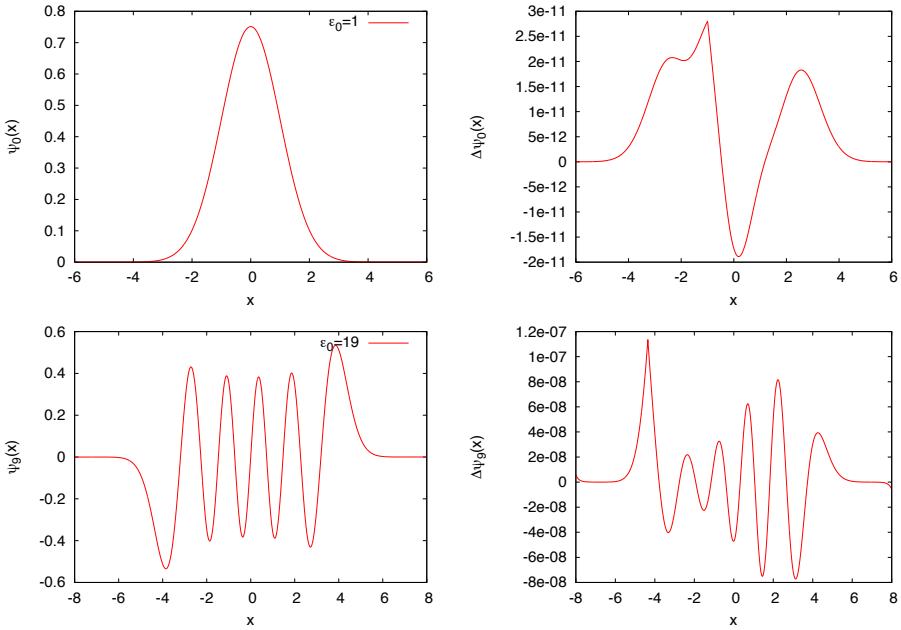



Figure 10.7: The eigenfunctions $\psi_0(x)$, $\psi_9(x)$ calculated by the program in `sch.f90`, `schHOC.f90`. The plot to the right shows the difference of the results from the known values (10.38).

Next, the calculation is repeated for the anharmonic oscillator for $\lambda = 0.5, 2.0$. We copy the file `schHOC.f90` to `schUOC.f90` and change the potential in the function `V(x)`:

```
!==========================================================
!file: schUOC.f
!...................
!------ potential:
real(8) function V(x)
 implicit none
 real(8) :: x, lambda
 lambda = 2.0D0
 V = x*x+2.0D0*lambda*x*x*x*x
end function V
....................
```

The wave functions are plotted in figure 10.8. We see that by increasing $\lambda$ the particle becomes more confined in space as expected. In table 10.3 we list the values of the energy $\epsilon_n$ for $n = 0, ..., 9$. By increasing $\lambda$, $\epsilon_n(\lambda)$ is increased. Table 10.4 lists the expectation values $\langle x^2 \rangle$, $\langle p^2 \rangle$ and $\Delta x \cdot \Delta p$ for

| $n$ | $\langle x^2 \rangle$ | $\langle p^2 \rangle$ | $\Delta x \cdot \Delta p$ |
|---|---|---|---|
| 0 | 0.500000000 | 0.4999977 | 0.4999989 |
| 1 | 1.500000284 | 1.4999883 | 1.4999943 |
| 2 | 2.499999747 | 2.4999711 | 2.4999854 |
| 3 | 3.499999676 | 3.4999441 | 3.4999719 |
| 4 | 4.499999607 | 4.4999082 | 4.4999539 |
| 5 | 5.499999520 | 5.4998633 | 5.4999314 |
| 6 | 6.499999060 | 6.4998098 | 6.4999044 |
| 7 | 7.499999642 | 7.4995484 | 7.4997740 |
| 8 | 8.499999715 | 8.4994203 | 8.4997100 |
| 9 | 9.499999837 | 9.4992762 | 9.4996380 |
| 10 | 10.500000012 | 10.4991160 | 10.4995580 |
| 11 | 11.499999542 | 11.4994042 | 11.4997019 |
| 12 | 12.499999610 | 12.4992961 | 12.4996479 |
| 13 | 13.499999705 | 13.4991791 | 13.4995894 |
| 14 | 14.499999835 | 14.4990529 | 14.4995264 |

Table 10.2: The expectation values $\langle x^2 \rangle$, $\langle p^2 \rangle$ and the product $\Delta x \cdot \Delta p$ for the simple harmonic oscillator for the states $|n\rangle$, $n = 0, \ldots, 14$.

| $n$ | $\epsilon_n$ | $\epsilon_{n,\lambda=0.5}$ | $\epsilon_{n,\lambda=2.0}$ |
|---|---|---|---|
| 0 | 1.0000 | 1.3924 | 1.9031 |
| 1 | 3.0000 | 4.6488 | 6.5857 |
| 2 | 5.0000 | 8.6550 | 12.6078 |
| 3 | 7.0000 | 13.1568 | 19.4546 |
| 4 | 9.0000 | 18.0576 | 26.9626 |
| 5 | 11.0000 | 23.2974 | 35.0283 |
| 6 | 13.0000 | 28.8353 | 43.5819 |
| 7 | 15.0000 | 34.6408 | 52.5723 |
| 8 | 17.0000 | 40.6904 | 61.9598 |
| 9 | 19.0000 | 46.9650 | 71.7129 |

Table 10.3: The values of the energy $\epsilon_n$ for the harmonic and anharmonic oscillator for $\lambda = 0.5, 2.0$. The values of the corresponding energy levels are increased with increasing $\lambda$.

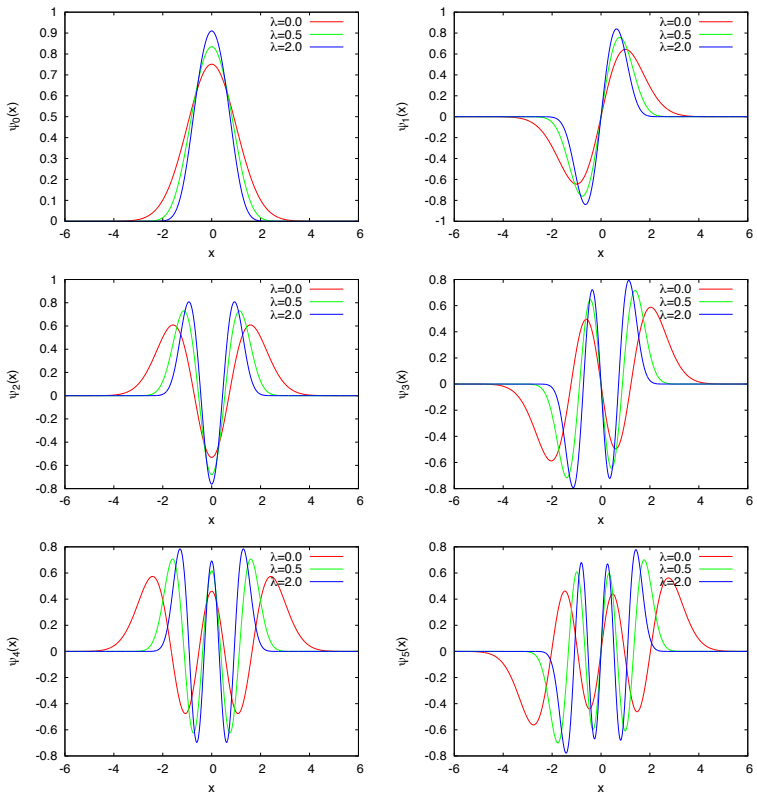



Figure 10.8: The wave functions of the anharmonic oscillator $\psi_{n,\lambda}(x)$ for $n = 0, 1, 2, 3, 4, 5$ and $\lambda = 0.5, 2.0$ compared to the respective ones of the simple harmonic oscillator. Increasing $\lambda$ yields stronger confinement of the particle in space.

the anharmonic oscillator for the states $|n\rangle$, $n = 0, \ldots, 9$. By increasing $\lambda$, $\Delta x = \sqrt{\langle x^2 \rangle}$ is decreased and $\Delta p = \sqrt{\langle p^2 \rangle}$ is increased. The product of the uncertainties $\Delta x \cdot \Delta p$ seems to be quite close to the corresponding values for the harmonic oscillator. The results should be compared with the ones obtained in table 9.1 of chapter 9.

## 10.6 The Lennard–Jones Potential

The Lennard–Jones potential is a simple phenomenological model of the interaction between two neutral atoms in a diatomic molecule. This is given by

$$V(x) = 4V_0 \left\{ \left(\frac{\sigma}{x}\right)^{12} - \left(\frac{\sigma}{x}\right)^{6} \right\} . \tag{10.40}$$

| | $\lambda = 0.5$ | | | $\lambda = 2.0$ | | |
|---|---|---|---|---|---|---|
| $n$ | $\langle x^2 \rangle$ | $\langle p^2 \rangle$ | $\Delta x \cdot \Delta p$ | $\langle x^2 \rangle$ | $\langle p^2 \rangle$ | $\Delta x \cdot \Delta p$ |
| 0 | 0.3058 | 0.8263 | 0.5027 | 0.2122 | 1.1980 | 0.5042 |
| 1 | 0.8013 | 2.8321 | 1.5064 | 0.5408 | 4.2102 | 1.5089 |
| 2 | 1.1554 | 5.3848 | 2.4944 | 0.7612 | 8.1513 | 2.4909 |
| 3 | 1.4675 | 8.2819 | 3.4862 | 0.9582 | 12.6501 | 3.4816 |
| 4 | 1.7509 | 11.4545 | 4.4784 | 1.1370 | 17.5955 | 4.4728 |
| 5 | 2.0141 | 14.8599 | 5.4707 | 1.3029 | 22.9169 | 5.4643 |
| 6 | 2.2617 | 18.4691 | 6.4631 | 1.4590 | 28.5668 | 6.4560 |
| 7 | 2.4970 | 22.2607 | 7.4555 | 1.6074 | 34.5103 | 7.4478 |
| 8 | 2.7220 | 26.2184 | 8.4478 | 1.7492 | 40.7206 | 8.4397 |
| 9 | 2.9384 | 30.3289 | 9.4402 | 1.8856 | 47.1762 | 9.4316 |

Table 10.4: The expectation values $\langle x^2 \rangle$, $\langle p^2 \rangle$ and the product $\Delta x \cdot \Delta p$ for the anharmonic oscillator for the states $|n\rangle$, $n = 0, \ldots, 9$. Note the decrease of $\Delta x = \sqrt{\langle x^2 \rangle}$ and the increase of $\Delta p = \sqrt{\langle p^2 \rangle}$ with increasing $\lambda$. The uncertainty product $\Delta x \cdot \Delta p$ seems to take values close to the corresponding ones of the harmonic oscillator for both values of $\lambda$. Compare the results in this table with the ones in table 9.1.

The repulsive term describes the Pauli interaction due to the overlapping of the electron orbitals, whereas the attractive term describes the Van der Waals force. The first one dominates at short distances and the latter at long distances. We choose $L = \sigma$ in (10.13) and define $v_0 = 2m\sigma^2 V_0/\hbar^2$. Equation (10.40) becomes

$$v(x) = 4v_0 \left\{ \left(\frac{1}{x}\right)^{12} - \left(\frac{1}{x}\right)^{6} \right\} , \tag{10.41}$$

whereas the eigenvalues $\epsilon_n$ are related to the energy values $E_n$ by

$$\epsilon_n = 4v_0 \left(\frac{E_n}{V_0}\right) . \tag{10.42}$$

The plot of the potential is shown in figure 10.5 for $v_0 = 250$. The minimum is located at $x_m = 2^{1/6} \approx 1.12246$ and its value is $-v_0$. The code for this potential is in the file `schLJ.f90`. The necessary changes to the code discussed in the previous sections are listed below:

```
!=========================================================
!file: schLJ.f90 (Lennard-Jones)
!..................
```

| $n$ | $\epsilon_n$ | $\langle x \rangle$ | $\langle p \rangle$ | $\langle x^2 \rangle$ | $\langle p^2 \rangle$ | $\Delta x$ | $\Delta p$ | $\Delta x \cdot \Delta p$ |
|---|---|---|---|---|---|---|---|---|
| 0 | -173.637 | 1.186 | 1.0e-10 | 1.415 | 34.193 | 0.091 | 5.847 | 0.534 |
| 1 | -70.069 | 1.364 | 6.0e-11 | 1.893 | 56.832 | 0.178 | 7.539 | 1.338 |
| 2 | -18.191 | 1.699 | -4.5e-08 | 2.971 | 39.480 | 0.291 | 6.283 | 1.826 |
| 3 | -1.317 | 2.679 | -2.6e-08 | 7.586 | 9.985 | 0.638 | 3.160 | 2.016 |

Table 10.5: The results for the Lennard-Jones potential with $v_0 = 250$. We find 4 bound states.

```
!------ potential:
real(8) function V(x)
 implicit none
 real(8) :: x,V0

 V0 = 250.0D0
 V  = 4.0D0*V0*(1.0D0/x**12-1.0D0/x**6)

end function V
!------ boundary conditions:
subroutine &
 boundary(xmin,xmax,psixmin,psipxmin,psixmax,psipxmax)
 implicit none
 real(8) :: xmin,xmax,psixmin,psipxmin,psixmax,psipxmax,V
 real(8) :: energy
 common/params/energy
!------ Initial values at xmin and xmax
 psixmin    =  exp(-xmin*sqrt(DABS(energy-V(xmin))))
 psipxmin   =  sqrt(DABS(energy-V(xmin)))*psixmin
 psixmax    =  exp(-xmax*sqrt(DABS(energy-V(xmax))))
 psipxmax   = -sqrt(DABS(energy-V(xmax)))*psixmax
end subroutine boundary
.............................
```

For the integration we choose $v_0 = 250$ and `xmin = 0.7`, $4 <$`xmax` $< 10$. The results are plotted in figure 10.9. There are four bound states. The first two ones are quite confined within the potential well whereas the last ones begin to "spill" out of it. Table 10.5 lists the results. We observe that $\langle p \rangle = 0$ within the attained accuracy as expected for real, bound states[14].

[14]For $\psi(+\infty) = \psi(0) = 0$ and $\psi^*(x) = \psi(x)$ we have that $i\langle p \rangle/\hbar = \int_0^{+\infty} \psi(x)(d/dx)\psi(x)\,dx = -\int_0^{+\infty}(d/dx)\psi(x)\psi(x)\,dx = 0$.

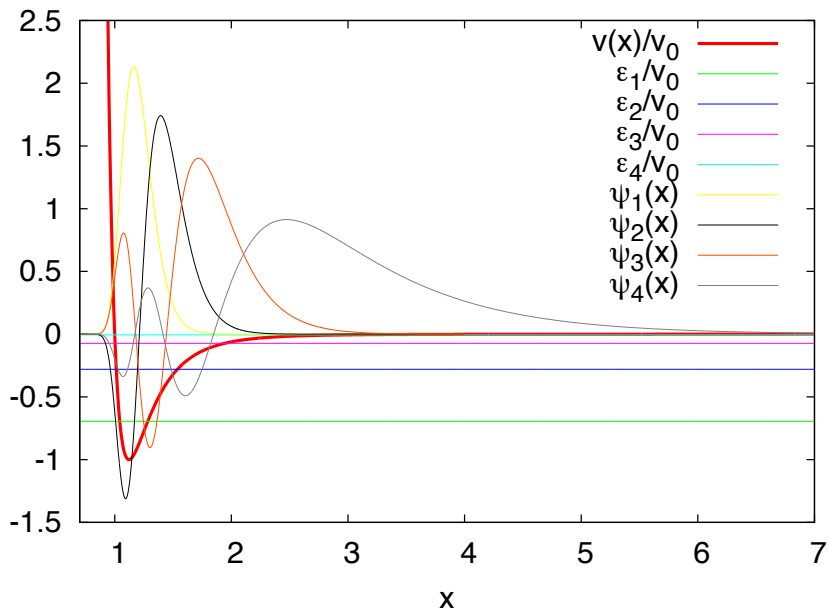



Figure 10.9: The four bound states for the Lennard-Jones potential with $v_0 = 250$. The bold red line is the potential $v(x)/v_0$. We plot the energy levels $\epsilon_n/v_0$ and the corresponding wave functions.

## 10.7 Problems

10.1 Add the necessary code to the program in the file `well.f90` so that the final wave function printed in the file `psi.dat` is properly normalized. The integral $\int_{-1}^{1} \psi(x)\psi(x)\,dx$ can be computed using the Simpson rule

$$\begin{aligned}\int_a^b f(x)\,dx &= (h/3)\,(f(x_0) + 4f(x_1) + 2f(x_2) + \ldots \\ &\quad +2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)\,.)\end{aligned}$$

The interval $[a, b]$ is discretized by $n$ points $x_0 = a, x_1, x_2, \ldots, x_n = b$ where $n$ is even. Each interval $[x_i, x_{i+1}]$ has width $h$.

10.2 Add the necessary code to the program in the file `well.f90` in order to calculate the number of nodes (zeroes) of the wave function. Using this result, the program should print the level $n$ of the calculated wave function $\psi_n(x)$.

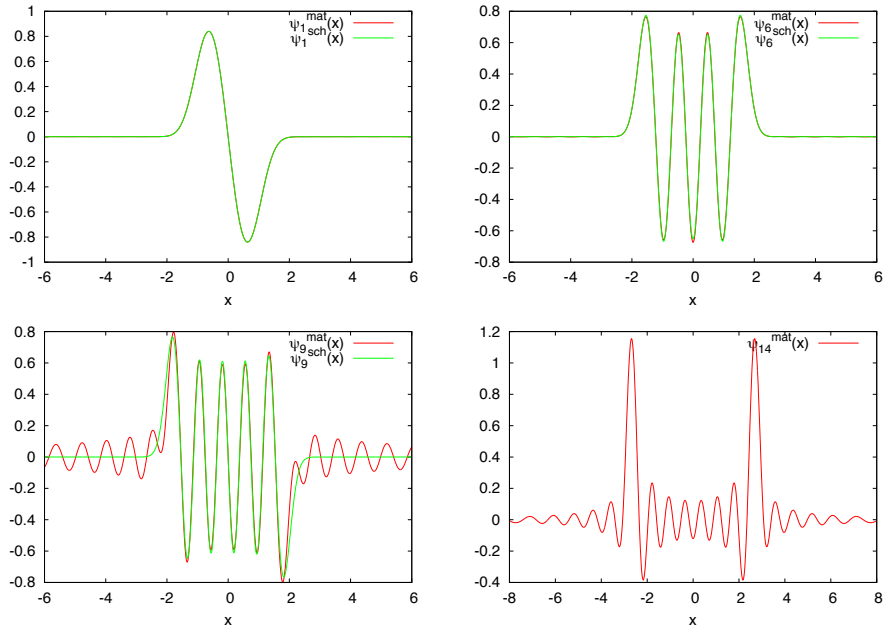



Figure 10.10: Comparison of the results of the calculation of the wave functions $\psi_{n,\lambda}(x)$ of the anharmonic oscillator for $\lambda = 2.0$ using the methods described in problem 12. The wave functions $\psi^{\text{sch}}(x)$ are the wave functions $\psi_{n,\lambda}(x)$ calculated using the methods described in this chapter. The wave functions $\psi^{\text{mat}}(x)$ are the wave functions $\psi_{n,\lambda}(x)$ calculated using the methods described in chapter 9 for Hilbert space dimension $N = 40$. Note the difference at large $x$. This is because the amplitudes $\psi_{n,\lambda}(x) = \langle x|n\rangle_\lambda$ for large $x$ receive contributions from states $|m\rangle$ with large $m$ (why?).

10.3 Calculate the wave functions of the energy eigenstates for the potential (10.27) with $v_0 < 0$. This is the problem of the (finite) potential well. Solve the problem for $v_0 = -100$ and $a = 0.3$. How many bound states do you find? Next study the influence of the wall on the solutions. Introduce a parameter $b$ so that $v(x \geq b) = +\infty$ and study the dependence of the solutions on $b$. Take $b = 0.35, 0.4, 0.5, 0.6, 0.8, 1.0, 1.5, 2.0, 2.5, 3.0$ and compute the difference of the first two energy eigenvalues. Estimate the accuracy of the method. Next lower the value of $|v_0|$ until there is no bound state. What is the relation between $a$ and $v_0$ when this happens? Compare with the analytic result which you know from your quantum mechanics course.
Hint: For the largest values of $b$, take `Nx > 1000`. When convergence is not achieved decrease `epsilon`.

10.4 Set $v_0 = 1000, 5000$ to the double well potential. Observe the (almost) degenerate states and plot the wave functions $\psi_{\pm,n} = (1/\sqrt{2})(\psi_n(x) \pm \psi_{n+1}(x))$, where $n$ is odd. Compare the results with the corresponding energy levels and eigenfunctions of the infinite square well. Increase $v_0$ to the point where you cannot solve the problem numerically.
Hint: For large $v_0$ the numerical effort is increased. For $|x| < a$ the wave function is almost zero and it is hard to obtain the non trivial wave function for $a < |x| < 1$. As the accuracy deteriorates, you should increase `epsilon` in the program so that convergence is achieved relatively fast.

10.5 Repeat problems 3 and 4 using the program `sch.f90`. Compare the results.

10.6 Study the bound states in the potentials

$$v(x) = \begin{cases} 0 & a < |x| \\ -V_0 & b < |x| < a \\ -V_1 & |x| < b \end{cases}$$

για $a = 1, b = 0.2, V_0 = 100, V_1 = 0, 50$ και

$$v(x) = \begin{cases} V_1 & x < 0 \\ -V_0 & 0 < x < a \\ 0 & a < x \end{cases}$$

για $a = 1, V_0 = 100, V_1 = +\infty, 10, 100$ και

$$v(x) = \begin{cases} V_1 & a < |x| \\ -V_0 & b < |x| < a \\ 0 & c < |x| < b \\ -V_0 & |x| < c \end{cases}$$

for $a = 1, b = 0.7, c = 0.6, 0.3, V_0 = 100, V_1 = +\infty, 10, 0$. In each case calculate $\langle x \rangle$, $\langle x^2 \rangle$, $\langle p \rangle$, $\langle p^2 \rangle$, $\Delta x$, $\Delta p$, $\Delta x \cdot \Delta p$.

10.7 Write a program that calculates the probability that a particle is found in an interval $[x_1, x_2]$ given the wave function calculated by the program in the file `sch.f90`. Apply your program on the results of the previous problem and calculate the intervals $[-x_1, x_1]$ where the probability to find the particle inside them is equal to $1/3$.

10.8 Fill the tables 10.3 and 10.4 with the results for $\lambda = 0.2$, $0.7$, $1.0$, $1.3$, $1.6$, $2.5$, $3.0$ and plot each expectation value as a function of $\lambda$.

10.9 A particle is under the influence of a potential

$$V(x) = \frac{\hbar^2}{2m}\alpha^2\lambda(\lambda-1)\left\{\frac{1}{2} - \frac{1}{\cosh^2(\alpha x)}\right\} .$$

The energy spectrum is given by

$$E_n = \frac{\hbar^2}{2m}\alpha^2\left\{\frac{\lambda(\lambda-1)}{2} - (\lambda-1-n)^2\right\}$$

for the values of $n = 0, 1, 2, \ldots$ for which $E_n > V_{\text{min}}$. Calculate the energy levels $\epsilon_n$ of the bound states numerically by setting $L = 1/\alpha$ in equation (10.13) and $\lambda = 4$. Plot the potential $v(x)$ and the corresponding eigenfunctions. Calculate the expectation values of the position and momentum, the uncertainties in position and momentum and their product. Repeat for $\lambda = 2, 6, 8, 10$.

10.10 Write a program that reads in a wavefunction and calculates the expectation value of the Hamiltonian

$$\langle\hat{H}\rangle = \int_{-\infty}^{+\infty}\psi(x)\left(-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x)\right)\psi(x)\,dx\,,$$

by assuming that $\psi(x)$ is real. Calculate $\psi_n(x)$ for the harmonic oscillator for $n = 1, \ldots, 10$ and show (numerically) that $\langle\hat{H}\rangle_n = E_n$.

10.11 Consider a particle in the Morse potential

$$V(x) = D_e\left\{\left(1 - \mathrm{e}^{-a(r-r_e)}\right)^2 - 1\right\} .$$

Calculate the energy spectrum of the bound states. Choose $L = 1/a$, $x = ar$, $x_e = ar_e$, $\lambda^2 = 2mD_e/a^2\hbar^2$ and obtain

$$v(x) = \lambda^2\left(\mathrm{e}^{-2(x-x_e)} - 2\mathrm{e}^{-(x-x_e)}\right) .$$

Compare your results with the known analytic solutions

$$\epsilon_n = \left(\lambda - n - \frac{1}{2}\right)^2$$

$$\psi_n(z) = N_n z^{\lambda-n-1/2}\mathrm{e}^{-z/2}L_n^{2\lambda-2n-1}(z)$$

where $z = 2\lambda\mathrm{e}^{-(x-x_e)}$, $N_n = n!\sqrt{(2\lambda-2n-1)/(\Gamma(n+1)\Gamma(2\lambda-n))}$, and $L_n^\alpha(z)$ is a Laguerre polynomial given by $L_n^\alpha(z) = (z^{-\alpha}\mathrm{e}^z/n!)(d^n/dz^n)(z^{n+\alpha}\mathrm{e}^{-z})$ $= (\Gamma(\alpha+2)/(\Gamma(n+2)\Gamma(\alpha-n+2))_1F_1(-n, \alpha+1, z)$. You can take $\lambda = 4$, $x_e = 1$ and calculate $\langle x\rangle$, $\langle x^2\rangle$, $\langle p\rangle$, $\langle p^2\rangle$, $\Delta x$, $\Delta p$, $\Delta x \cdot \Delta p$.

10.12 Calculate the wave functions of the eigenstates of the Hamiltonian for the anharmonic oscillator for $\lambda = 2.0$ and $n = 0, \ldots, 15$. Calculate the wavefunctions using the program `anharmonic.f90` of chapter 9 for $N = 15, 40, 100$ and compare the two results.
Hint: Write a program that calculates the energy eigenfunctions of the simple harmonic oscillator

$$\psi_n(x) = \frac{1}{\sqrt{2^n n! \sqrt{\pi}}} \mathrm{e}^{-x^2/2} H_n(x)$$

where the Hermite polynomials satisfy the relations

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x), \quad H_0(x) = 1, \quad H_1(x) = 2x\,.$$

The program `anharmonic.f90` calculates the eigenstates of the anharmonic oscillator

$$|n\rangle_\lambda = \sum_{m=0}^{N-1} \mathtt{H(m+1,n+1)}\,|m\rangle$$

by storing the coefficients of the linear expansion in the elements of the array `H(N,N)`. The same relation holds for the corresponding wave functions $\psi_{n,\lambda}(x)$, $\psi_n(x)$. From $\psi_n(x)$ and `H(i,j)` calculate $\psi_{n,\lambda}(x)$ for $-8 < x < 8$ and determine the accuracy achieved by the calculation for each $N$. For which values of $x$ do you obtain large discrepancies between your results? Remember that for large $x$, the states of high energy contribute more than for small $x$. Figure 10.10 can help you understanding this statement.

# Chapter 11

# The Random Walker

In this chapter we will study the typical path followed by a ... drunk when he decides to start walking from a given position. Because of his drunkenness, his steps are in random directions and uncorrelated. These are the basic properties of the models that we are going to study. These models are related to specific physical problems like the Brownian motion, the diffusion, the motion of impurities in a lattice, the large distance properties of macromolecules etc. In the physics of elementary particles random walks describe the propagation of free scalar particles and they most clearly arise in the Feynman path integral formulation of the euclidean quantum field theory. Random walks are precursors to the theory of random surfaces which is related to the theory of two dimensional "soft matter" membranes, two dimensional quantum gravity and string theory [41].

The geometry of a typical path of a simple random walk is not classical and this can be seen from two of its non classical properties. First, the average distance traveled by the random walker is proportional to the *square root* of the time traveled, i.e. the classical relation $r = vt$ does not apply. Second, the geometry of the path of the random walker has fractal dimension which is larger than one[1]. Similar structures arise in the study of quantum field theories and random surfaces, where the non classical properties of a typical configuration can be described by appropriate generalizations of these concepts. For further study we refer to [7, 40, 41, 42].

In order to simulate a stochastic system on the computer, it is necessary to use random number generators. In most of the cases, these are deterministic algorithms that generate a sequence of *pseudorandom* num-

[1]More precisely, the Hausdorff dimension of the simple random walk is $d_H = 2$.

bers distributed according to a desired distribution. The heart of these algorithms generate numbers distributed uniformly from which we can generate any other complex distribution. In this chapter we will study simple random number generators and learn how to use high quality, research grade, portable, random number generators.

## 11.1 (Pseudo)Random Numbers

The production of pseudorandom[2] numbers is at the heart of a Monte Carlo simulation. The algorithm used in their production is deterministic: The generator is put in an initial state and the sequence of pseudorandom numbers is produced during its "time evolution". The next number in the sequence is determined from the current state of the generator and it is in this sense that the generator is deterministic. Same initial conditions result in exactly the same sequence of pseudorandom numbers. But the "time evolution" is chaotic and "neighboring" initial states result in very different, uncorrelated, sequences. The chaotic properties of the generators is the key to the pseudorandomness of the numbers in the sequence: the numbers in the sequence decorrelate exponentially fast with "time". But this is also the weak point of the pseudorandom number generators. Bad generators introduce subtle correlations which produce systematic errors. Truly random numbers (useful in cryptography) can be generated by using special devices based on e.g. radioactive decay or atmospheric noise[3]. Almost random numbers are produced by the special files `/dev/random` and `/dev/urandom` available on unix systems, which read bits from an entropy pool made up from several external sources (computer temperature, device noise etc).

Pseudorandom number generators, however, are the source of random numbers of choice when *efficiency* is important. The most popular generators are the *modulo* generators (D.H. Lehmer, 1951) because of their simplicity. Their state is determined by only one integer $x_{i-1}$ from which the next one $x_i$ is generated by the relation

$$x_i = a\, x_{i-1} + c \,(\mathrm{mod}\, m) \tag{11.1}$$

[2]We can't define what a random process is, only what it isn't. Outcomes which lack discernable patterns are assumed to be random. If there is no way to predict an event, we say it is random...Thus, there is no definition of what randomness is, only definitions of what it isn't. See Chris Wetzel, "Can you behave randomly?", `http://faculty.rhodes.edu/wetzel/random/level23intro.html`.

[3]There are online services which provide such sequences like `www.random.org`, `www.fourmilab.ch/hotbits/` and others.

for appropriately chosen values of $a$, $c$ and $m$. In the bibliography, there is a lot of discussion on the good and bad choices of $a$, $c$ and $m$, which depend on the programming language and whether we are on a 32–bit or 64–bit systems. For details see the chapter on random numbers in [8].

The value of the integer $m$ determines the maximum period of the sequence. It is obvious that if the sequence encounters the same number after $k$ steps, then the exact same sequence will be produced and $k$ will be the period of the sequence. Since there are at most $m$ different numbers, the period is at most equal to $m$. For a bad choice of $a$, $c$ and $m$ the period will be much smaller. But $m$ cannot be arbitrarily large since there is a maximum number of bits that computers use for the storage of integers. For 4-byte (32 bit) unsigned integers the maximum number is $2^{32} - 1$, whereas for signed integers $2^{31} - 1$. One can prove[4] that a good choice of $a$, $c$ and $m$ results in a sequence which is a permutation $\{\pi_1, \pi_2, \ldots, \pi_m\}$ of the numbers $1, 2, \ldots, m$. This is good enough for simple applications that require fast random number generation but for serious calculations one has to carefully balance efficiency with quality. Good quality random generators are more complicated algorithms and their states are determined by more than one integer. If you need the source code for such generators you may look in the bibliography, like in e.g. [4], [5], [8], [44]. If portability is an issue, we recommend the RANLUX random number generator [44] or the Marsaglia, Zaman and Tsang generator. The Fortran code for `RANLUX` can also be found in the accompanying software, whereas the MZT generator can be found in Berg's book/site [5].

In order to understand the use of random number generators, but also in order to get a feeling of the problems that may arise, we list the code of the two functions `naiveran()` and `drandom()`. The first one is obviously problematic and we will use it in order to study certain type of correlations that may exist in the generated sequences of random numbers. The second one is much better and can be used in non–trivial applications, like in the random walk generation or in the Ising model simulations studied in the following chapters.

The function `naiveran()` is a simple application of equation (11.1) with $a = 1277$, $c = 0$ and $m = 2^{17}$:

```
!==========================================
!File: naiveran.f90
!Program to demonstrate the usage of a modulo
```

[4] See Knuth [43].

```
!generator with a bad choice of constants
!resulting in strong pair correlations between
!generated numbers
!=============================================
real(8) function naiveran()
 implicit none
 integer            :: iran=13337
 common /naiveranpar/ iran
 integer,parameter :: m = 131072 ! equal to 2**17
 integer,parameter :: a = 1277

 iran = a*iran
 iran = MOD(iran,m)

 naiveran = iran/DBLE(m)

end function naiveran
```

The function `drandom()` is also an application of the same equation, but now we set $a = 7^5$, $c = 0$ and $m = 2^{31} - 1$. This is the choice of Lewis, Goodman και Miller (1969) and provides a generator that passes many tests and, more importantly, it has been used countless of times successfully. One technical problem is that, when we multiply $x_{i-1}$ by $a$, we may obtain a number which is outside the range of 4-byte integers and this will result in an "integer overflow". In order to have a fast and portable code, it is desirable to stay within the range of the $2^{31} - 1$ positive, 32-bit (4 byte), signed integers. Schrage has proposed to use the relation

$$(ax_{i-1}) \mod m = \begin{cases} a\,(x_{i-1} \mod q) - r\left[\frac{z}{q}\right] & \text{if it is} \geq 0 \\ a\,(x_{i-1} \mod q) - r\left[\frac{z}{q}\right] + m & \text{if it is} < 0 \end{cases} \tag{11.2}$$

where $m = aq + r$, $q = [m/a]$ and $r = m \mod a$. One can show that if $r < q$ and if $0 < x_{i-1} < m - 1$, then $0 \leq a(x_{i-1} \mod q) \leq m - 1$, $0 \leq r[z/q] \leq m - 1$ and that (11.2) is valid. The period of the generator is $2^{31} - 2 \approx 2 \times 10^9$. The proof of the above statements is left as an exercise to the reader.

```
!=====================================================
!File: drandom.f90
!Implementation of the Schrage algorithm for a
!portable modulo generator for 32 bit signed integers
!(from numerical recipes)
!
```

```
!returns uniformly distributed pseudorandom numbers
! 0.0 < x < 1.0 (0 and 1 excluded)
!=================================================
real(8) function drandom()
 implicit none
 integer,parameter :: a = 16807       ! a = 7**5
 integer,parameter :: m = 2147483647 ! m = a*q+r = 2**31-1
 integer,parameter :: q = 127773      ! q = [m/a]
 integer,parameter :: r = 2836        ! r = MOD(m,a)
 real(8),parameter :: f = (1.0D0/m)
 integer           :: p
 integer           :: seed
 real(8)           :: dr
 common /randoms/seed

101 continue
 p       = seed/q                 !  = [seed/q]
 seed    = a*(seed- q*p) - r*p !  = a*MOD(seed,q)-r*[seed/q]
 if(seed .lt. 0) seed = seed + m
 dr      = f*seed
 if( dr .le. 0.0D0 .or. dr .ge. 1.0D0) goto 101
 drandom = dr
end function drandom
```

The line that checks the result produced by the generator is necessary in order to check for the number 0 which appears once in the sequence. This adds a $10-20\%$ overhead, depending on the compiler. If you don't care about that, you may remove the line. Note that the number `seed` is put in a common block so it can be accessed by other parts of the program.

Now we will write a program in order to test the problem of correlations in the sequence of numbers produced by `naiveran()`. The program will produce pairs of integers $(i, j)$, where $0 \leq i, j < 10000$, which are subsequently mapped on the plane. This is done by taking the integer part of the numbers $L\,u$ with $L = 10000$ and $0 \leq u < 1$ is the random number produced by the generator:

```
!=========================================================
!Program that produces N random points (i,j) with
!0<= i,j < 10000. Simple qualitative test of  serial
!correlations of random number generators on the plane.
!
!compile:
!gfortran correlations2ran.f90 naiveran.f90 drandom.f90
!=========================================================
program correlations2
```

```
 implicit none
 integer,parameter :: L = 10000
 integer           :: i,N
 character(10)     :: arg
 real(8)           :: naiveran,drandom
 integer           :: seed
 common /randoms/     seed
!Read the number of points from first command argument
 if(IARGC() .EQ. 1)then
  call GETARG(1,arg); read(arg,*)N !convert string->integer
 else !default value, if no N given by user:
  N=1000
 endif
 seed = 348325
 do i=1,N
  print *,INT(L * naiveran()),INT(L * naiveran())
! print *,INT(L * drandom ()),INT(L * drandom ())
 enddo

end program correlations2
```

The program can be found in the file `correlations2ran.f90`. In order to test `naiveran()` we compile with the command

```
> gfortran correlations2ran.f90 naiveran.f90 -o naiveran
```

whereas in order to test `drandom()` we uncomment the `print` lines as follows

```
! print *,INT(L * naiveran()),INT(L * naiveran())
  print *,INT(L * drandom ()),INT(L * drandom ())
```

and recompile:

```
> gfortran correlations2ran.f90 drandom.f90 -o drandom
```

These commands result in two executable files `naiveran` and `drandom`. In order to see the results we run the commands

```
> ./naiveran 100000 > naiveran.out
> ./drandom  100000 > drandom.out
> gnuplot
gnuplot> plot "naiveran.out" using 1:2 with dots
gnuplot> plot "drandom.out"  using 1:2 with dots
```

which produce $10^5$ points used in the plots in figures 11.1 and 11.2. In the plot of figure 11.1, we see the pair correlations between the numbers produced by `naiveran()`. Figure 11.2 shows the points produced by `drandom()`, and we can see that the correlations shown in figure 11.1 have vanished. The plot in figure 11.2 is qualitative, and a detailed, quantitative, study of `drandom()` shows that the pairs $(u_i, u_{i+1})$ that it produces, do not pass the $\chi^2$ test when we have more than $10^7$ points, which is much less than the period of the generator. In order to avoid such problems, there are many solutions that have been proposed and the simplest among them "shuffle" the results so that the low order serial correlations vanish. Such generators will be discussed in the next section. The uniform distribution of the random numbers produced

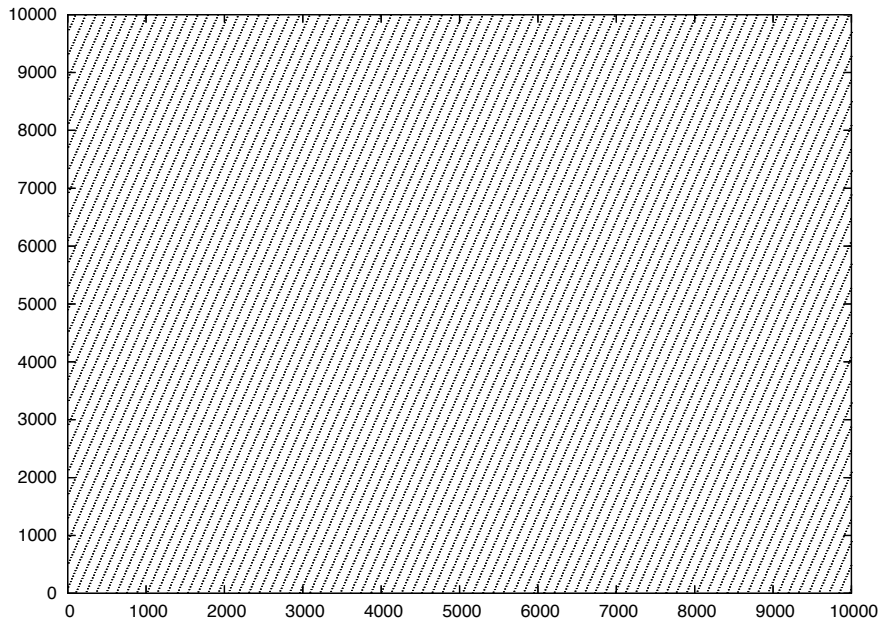



Figure 11.1: Pairs of pseudorandom numbers produced by the function `naiveran()`. The correlations among pairs of such numbers show in the distribution of such pairs on a clearly seen lattice.

can be examined graphically by constructing a histogram of the relative frequency of their appearance. In order to construct the histograms we use the script `histogram` which is written in the awk language[5] as shown

[5]See the accompanying software in the `Tools` directory. Give the command `histogram -- -h` which prints short usage instructions. I hope you remember how to make the file `histogram` executable and put it in your path...

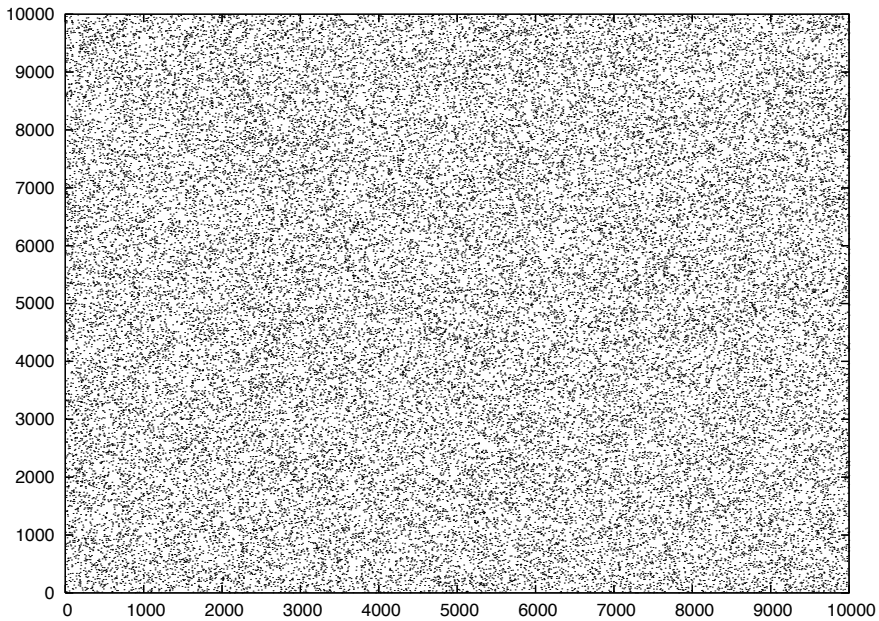



Figure 11.2: Pairs of pseudorandom numbers produced by the function `drandom()`. These points have a random distribution on the plane compared to those generated by `naiveran()`.

below:

```
> histogram -v f=0.01 drandom.out > drandom.hst
> gnuplot
gnuplot> plot "drandom.hst" using 1:3 with histeps
gnuplot> plot [:][0:] "drandom.hst" using 1:3 with histeps
```

The command `histogram -v f=0.01` constructs a histogram of the data so that the bin width is $1/0.01 = 100$. The reciprocal of the number following the option `-v f=0.01` defines the bin width. The histogram is saved in the file `drandom.out`.

The results are shown in figures 11.3 and 11.4. Next, we study the variance of the measurements, shown in figure 11.3. The variance is decreased with the size of the sample of the collected random numbers. This is seen in the histogram of figure 11.5. For a quantitative study of the dependence of the variance on the size $n$ of the sample, we calculate

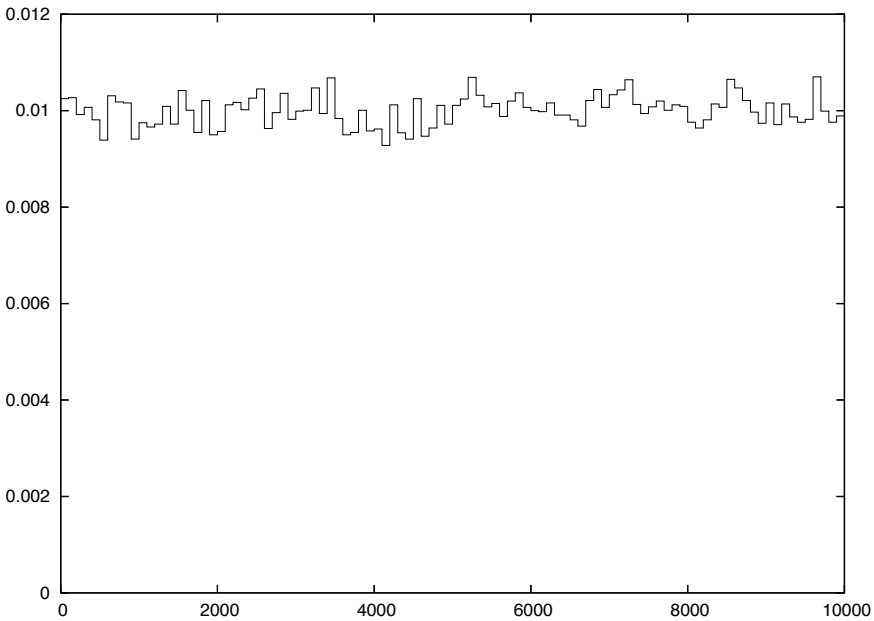



Figure 11.3: The relative frequency distribution of the pseudorandom numbers generated by `drandom()`. The distribution is uniform within $(0,1)$ and we see the deviations from the average value.

the standard deviation

$$\sigma = \sqrt{\frac{1}{n-1}\left(\frac{1}{n}\sum_{i=1}^{n} x_i^2 - \left(\frac{1}{n}\sum_{i=1}^{n} x_i\right)^2\right)}\,, \tag{11.3}$$

where $\{x_i\}$ is the sequence of random numbers. Figure 11.6 plots this relation. By fitting

$$\ln\sigma \sim \frac{1}{2}\ln(n)\,, \tag{11.4}$$

to a straight line, we see that

$$\sigma \sim \frac{1}{\sqrt{n}}\,. \tag{11.5}$$

If we need to generate random numbers which are distributed according to the probability density $f(x)$ we can use a sequence of uniformly distributed random numbers in the interval $(0,1)$ as follows: Consider the cumulative distribution function

$$0 \le u \equiv F(x) = \int_{-\infty}^{x} f(x')\,dx' \le 1\,, \tag{11.6}$$

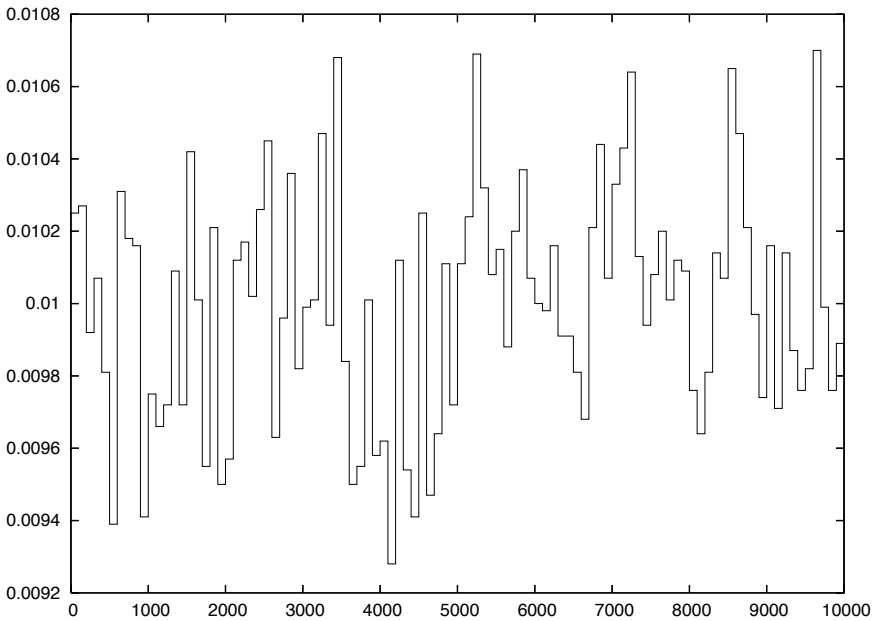



Figure 11.4: Same as in figure 11.3, but with the scale enlarged, so that the dispersion of the histogram values is clearly seen.

which is equal to the area under the curve $f(x)$ in the interval $(-\infty, x]$ and it is equal to the probability $P(x' < x)$. If $u$ is uniformly distributed in the interval $(0, 1)$ then we have that $P(u' < u) = u$. Therefore $x = F^{-1}(u)$ is such that $P(x' < x) = u = F(x)$ and follows the $f(x)$ distribution. Therefore, if $u_i$ form a sequence of uniformly distributed random numbers, then the numbers

$$x_i = F^{-1}(u_i) \tag{11.7}$$

form a sequence of random numbers distributed according to $f(x)$.

Consider for example the Cauchy distribution

$$f(x) = \frac{1}{\pi} \frac{c}{c^2 + x^2} \quad c > 0\,. \tag{11.8}$$

Then

$$F(x) = \int_{-\infty}^{x} f(x')\, dx' = \frac{1}{2} + \frac{1}{\pi} \tan^{-1}\left(\frac{x}{c}\right)\,. \tag{11.9}$$

According to the previous discussion, the random number generator is given by the equation

$$x_i = c \tan\left(\pi u_i - \pi/2\right) \tag{11.10}$$

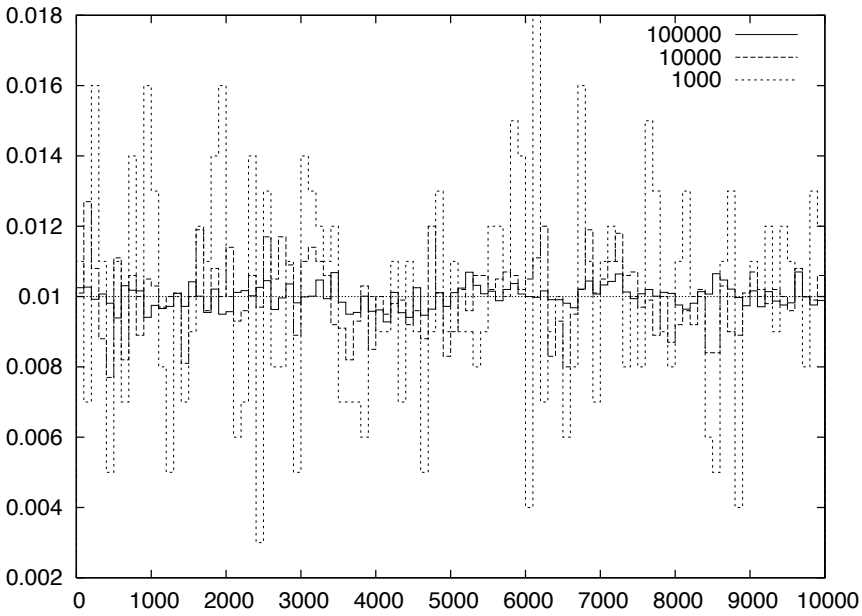



Figure 11.5: The relative frequency distribution of the pseudorandom numbers generated by `drandom()` as a function of the sample size $n$ for $n = 1000, 10000, 100000$.

or equivalently (for a more efficient generation)

$$x_i = c \tan(2\pi u_i) \ . \tag{11.11}$$

The generator of Gaussian random numbers is found in many applications. The Gaussian distribution is given by the probability density

$$g(x) = \frac{1}{\sqrt{2\pi}\,\sigma} \mathrm{e}^{-x^2/(2\sigma^2)} \tag{11.12}$$

The cumulative distribution function is

$$G(x) = \int_{-\infty}^{x} g(x')\,dx' = \frac{1}{2} + \frac{1}{2}\mathrm{erf}\left(\frac{x}{\sqrt{2}\,\sigma}\right) \tag{11.13}$$

where $\mathrm{erf}(x) = \int_{-\infty}^{x} \exp\{-(x')^2\}\,dx'$ is the error function. The error function, as well as its inverse, can be calculated numerically, but this would result in a slow computation. A trick to make a more efficient calculation is to consider the probability density $\rho(x, y)$ of *two* independent Gaussian random variables $x$ and $y$

$$\rho(x,y)\,dx\,dy = \frac{1}{\sqrt{2\pi}\,\sigma}\mathrm{e}^{-x^2/(2\sigma^2)}\ \frac{1}{\sqrt{2\pi}\,\sigma}\mathrm{e}^{-y^2/(2\sigma^2)}\,dx\,dy = \frac{1}{2\pi\sigma^2}\mathrm{e}^{-r^2/(2\sigma^2)}\,r\,dr\,d\phi \tag{11.14}$$

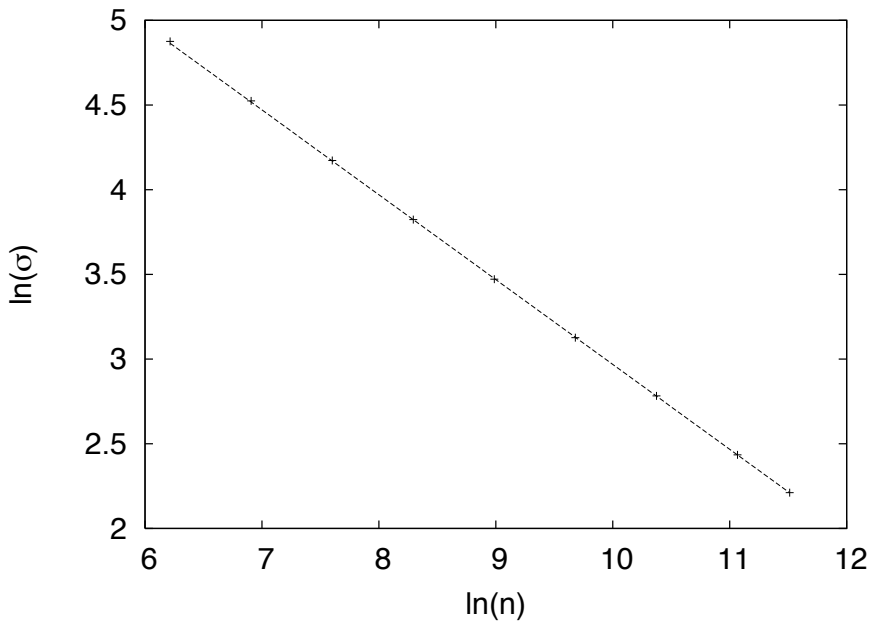



Figure 11.6: The dependence of the variance (11.3) on $n$ for the distribution of random numbers generated by `drandom()`.

where $x = r\cos\phi$, $y = r\sin\phi$. Then we have that

$$u = G(r) = \int_0^r \int_0^{2\pi} dr\, d\phi\, \rho(r,\phi) = 1 - \mathrm{e}^{-r^2/(2\sigma^2)}\,, \tag{11.15}$$

which, upon inversion, it gives

$$r = \sigma\sqrt{-2\,\ln(1-u)}\,. \tag{11.16}$$

Therefore it is sufficient to generate a sequence $\{u_i\}$ of uniformly distributed random numbers and take

$$r_i = \sigma\sqrt{-2\,\ln(u_i)} \tag{11.17}$$

$$\phi_i = 2\pi u_{i+1} \tag{11.18}$$

$$x_i = r_i\cos\phi_i \tag{11.19}$$

$$x_{i+1} = r_i\sin\phi_i\,. \tag{11.20}$$

The algorithm shown above gives a sequence of pseudorandom numbers $\{x_i\}$, which follow the Gaussian distribution[6]. The program for $\sigma = 1$ is listed below:

[6]It can be shown that $x_i$, $x_{i+1}$ are statistically independent.

```
!=================================================
!Function to produce random numbers distributed
!according to the gaussian distribution
!g(x) = 1/(sigma*sqrt(2*pi))*exp(-x**2/(2*sigma**2))
!=================================================
real(8) function gaussran()
 implicit none
 real(8),parameter :: sigma = 1.0D0
 real(8)           :: r,phi
 logical,save      :: new   = .TRUE.
 real(8),save      :: x
 real(8),parameter :: PI2   = 6.28318530717958648D0
 real(8)           :: drandom
 if(new)then
  new      = .FALSE.
  r        =     drandom()
  phi      = PI2*drandom()
  r        = sigma*sqrt(-2.0D0*log(r))
  x        = r*cos(phi)
  gaussran = r*sin(phi)
 else
  new      = .TRUE.
  gaussran = x
 endif
end function gaussran
```

The result is shown in figure 11.7. Notice the `SAVE` attribute for the variables `new` and `x`. This means that their values are saved between calls of `drandom`. We do this because each time we calculate according to (11.17), we generate two random numbers, whereas the function returns only one. The function needs to know whether it is necessary to generate a new pair $(x_i, x_{i+1})$ (this is what the "flag" `flag` marks) and, if not, to return the previously generated number, saved in the variable `x`. The analysis of the results is left as an exercise to the reader.

## 11.2 Using Pseudorandom Number Generators

The function `drandom()` is good enough for the problems studied in this book. However, in many demanding and high accuracy calculations, it is necessary to use higher quality random numbers and/or have the need of much longer periods. In this section we will discuss how to use two high quality, efficient and portable generators which are popular among many researchers.

The first one is an intrinsic procedure in the Fortran language, the

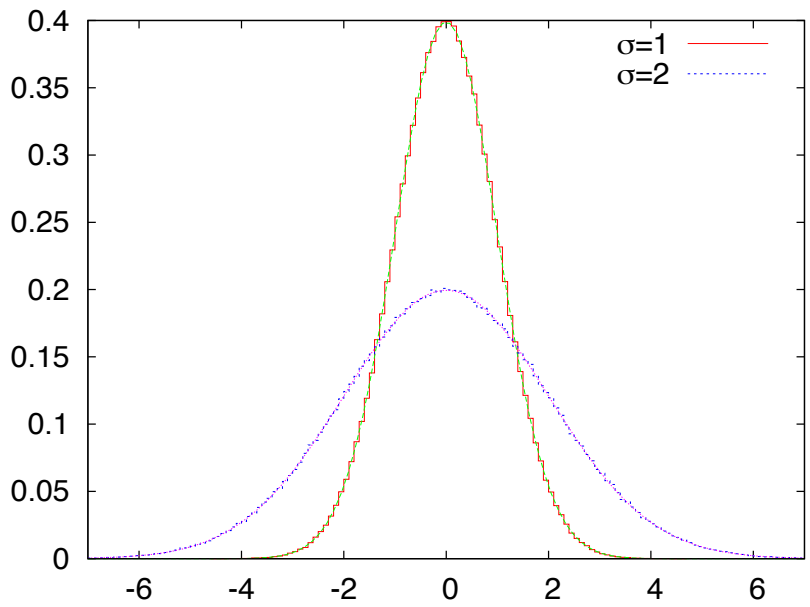



Figure 11.7: The distribution of pseudorandom numbers generated by `gaussran()` for $\sigma = 1$ and $\sigma = 2$. The histogram is superimposed to the plot of (11.12).

subroutine `RANDOM_NUMBER`. The algorithm implemented is not universal and depends on the Fortran environment[7]. For the `gfortran` compiler `RANDOM_NUMBER` uses the "multiply-with-carry" algorithm of George Marsaglia in combination with a modulo and shift-register generator. The period is larger than $2^{123} \approx 10^{37}$. The state of the generator is determined by more than one numbers. In order to use it we should learn

- how to start from a new state
- how to save the current state
- how to restart from a previously saved state
- and, of course, how to obtain the random numbers.

Saving the current state of the generator is very important when we execute a job that is split in several parts (checkpointing). This is done very often on computer systems that set time limits for jobs or when our

[7]Read carefully the documentation of your compiler. For this reason, the number `NSEEDS` can be different among different implementations of Fortran.

jobs are so long (more than 8-10 hours) that it will be painful to loose the resources (time and money) spent for the calculation in case of a computer crash. If we want to restart the job from exactly the same state as it was before we stopped, we also need to restart the random number generator from the same state.

Starting from a new, fresh state is called seeding. The seeding of `RANDOM_NUMBER` is done by an unspecified number of `NSEEDS` integers. In order to get this number, we should call the subroutine `RANDOM_SEED(size = NSEEDS)` which returns the number of seeds `NSEEDS` when its argument is `size = NSEEDS`. Then we have to define the integer values of the array `seeds(NSEEDS)` and call again the routine `RANDOM_SEED(PUT = seeds)` with the argument `PUT = seeds`, which will seed the generator from the seeds in `seeds`. In the code listed below we show how to seed the generator by using only one integer `seed`:

```
integer                 :: NSEEDS
integer,allocatable     :: seeds(:)
integer                 :: seed
!-----------------------------------------------
seed = 47279823
call RANDOM_SEED(size = NSEEDS)
ALLOCATE(seeds(NSEEDS))
seeds = seed + 37 * (/ (i - 1, i = 1, NSEEDS) /)
call RANDOM_SEED(PUT = seeds)
```

The last line[8] uses the values stored in the array `seeds(1) ... seeds(NSEEDS)` in order to initialize `RANDOM_NUMBER`. It is important to note that, using this method, the same `seed` will generate the same sequence of pseudo-random numbers.

Sometimes we need to initialize the random number generator from as a random initial state as possible, so that each time that we run our program, a different sequence of random numbers is generated. For Unix like systems, like the GNU/Linux system, we can use the two special files `/dev/random` and `/dev/urandom` in order to generate cryptographic-grade random numbers. These generate random bits from the current state of the computer and it is practically impossible to predict the obtained sequence of bits. It is preferable to use `/dev/urandom` because `/dev/random` ceases to work when there are no new random bits in its pool and waits until they are safely generated. The code that uses[9] `/dev/urandom` for

[8]The line before sets `seeds(1)=seed`, `seeds(2)=seed+37`, `seeds(3)=seed+37*2`, ...

[9]The `access='stream'` argument in `open` is a Fortran 2003 and above feature. Make sure that your compiler accepts it.

seeding is

```
open(unit=13, file='/dev/urandom', access='stream', &
 form='unformatted')
call RANDOM_SEED(size = NSEEDS)
ALLOCATE(seeds(NSEEDS))
read (13) seeds
close(13)
```

The special file `/dev/urandom` provides binary, non printable, data so it is necessary to open it with the `unformatted` option. For the same reason, the command `read` does not have a format instruction but only the unit number. It reads the number of bits necessary to fill the array `seeds`. If we need to work in an environment where the special file `/dev/urandom` is not available, it is possible to seed using the current time and the process number ID. The latter is necessary in case we start several processes in parallel and we need different seeds. Check the file `seed.f90` in order to see how to do it[10].

In order to save the current state of the random number generator use the subroutine `RANDOM_SEED(GET = seeds)` with argument `GET = seeds`. The call stores the necessary information in the array `seeds`. We can save the values of the array `seeds` in order to use them to restart the random number generator from exactly the same state. The necessary code is:

```
integer                 :: NSEEDS
integer,allocatable     :: seeds(:)
!-----------------------------------------
call RANDOM_SEED(size = NSEEDS)
ALLOCATE(seeds(NSEEDS))
call RANDOM_SEED(GET = seeds)
open(unit=11,file='state')
write(11,*)seeds
```

In order to restart the generator from a saved state, we read the values of the array `seeds` and call `RANDOM_SEED(PUT = seeds)` with argument `PUT = seeds`. The following code reads `seeds` from a file named `state`:

```
open(unit=11,file='state')
read(11,*)seeds
```

[10]You can also use the operating system in order to pass random seeds to your program. Try the commands `set x = `< /dev/urandom tr -dc "[:digit:]" | head -c9 | awk 'printf "%d",$1'` ; echo $x` and `set x = `perl -e 'srand();print int(100000000*rand());'` ; echo $x`. Use the value of the variable `x` for a seed.

```
 call RANDOM_SEED(PUT = seeds)
```

In order to generate random numbers, we can use a scalar variable and generate them one by one or we can use an array which RANDOM_NUMBER will fill with random numbers. The first method has a small overhead and in same cases we will prefer the second one. The code that applies the first method is

```
 real(8)                 :: r
!-----------------------------------------
 do icount = 1,10
  call RANDOM_NUMBER(r)
  print *,r
 enddo
```

and the code that applies the second is

```
 integer,parameter       :: NR=20
 real(8),dimension(NR)   :: randoms
 call RANDOM_NUMBER(randoms)
 print *,randoms
```

The code in the file test_random_number.f90 implements all of the above tasks and we list it below:

```
program use_random_number
 implicit none
 integer                 :: NSEEDS
 integer,allocatable     :: seeds(:)
 integer                 :: seed
 real(8)                 :: r
 integer,parameter       :: NR=20
 real(8),dimension(NR)   :: randoms
 integer(8)              :: icount
 integer                 :: i
!-----------------------------------------
!start from a new seed:
 seed = 47279823
!get number of seeds for generator:
 call RANDOM_SEED(size = NSEEDS)
 ALLOCATE(seeds(NSEEDS))
!fill in the rest of the seeds:
 seeds = seed + 37 * (/ (i - 1, i = 1, NSEEDS) /)
!initialize the generator from the arrays seeds:
 call RANDOM_SEED(PUT = seeds)
!-----------------------------------------
```

```
!generate random numbers one by one:
 do icount = 1,10
  call random_number(r)
  print *,r
 enddo
!generate random numbers in an array:
 call random_number(randoms)
 print '(1000G28.17)',randoms
!-----------------------------------------
!save state of random_number:
 open(unit=11,file='rannum.seed')
 call RANDOM_SEED(GET = seeds)
 write(11,'(5I20)')seeds
 close(11)
!-----------------------------------------
!generate some randoms:
 call random_number(randoms)
 print '(A,1000G28.17)','#FIRST :',randoms
!-----------------------------------------
!read state of random_number:
 open(unit=11,file='rannum.seed')
 read(11,*)seeds
 call RANDOM_SEED(PUT = seeds)
!-----------------------------------------
!generate same randoms:
 call random_number(randoms)
 print '(A,1000G28.17)','#SECOND:',randoms
end program use_random_number
```

Use the following commands in order to compile and see the results:

```
> gfortran test_random_number.f90 -o random_number
> ./random_number
```

A very high quality, portable random number generator was proposed by Martin Lüscher [44], and a program that implements it is Ranlux. Besides the high quality random numbers and a period greater than $10^{171}$, a great advantage of RANLUX is that it will run in any Fortran environment. The code, which can also be found in the accompanying software, has been written by Fred James and you can download it in its original form from the links given in the bibliography [44]. The generator uses a subtract-with-borrow algorithm by Marsaglia and Zaman [46], which has a very large period but fails some of the statistical tests. Based on the chaotic properties of the algorithm, Lüscher attributed the problems to short time autocorrelations and proposed a solution in order to eliminate them.

In order to start RANLUX using a single seed, use the subroutine RLUXGO. The necessary code is:

```
integer                    :: seed,ranlux_level
!-------------------------------------------
seed         = 58266273
ranlux_level = 2
call RLUXGO(ranlux_level,seed,0,0)
```

The value of the variable ranlux_level determines the quality of random numbers and it can take the values 1, 2, 3 or 4. Setting ranlux_level=2 is enough for the needs of this book and ranlux_level=3 is the default value. A larger value of ranlux_level requires more computational effort (see problem).

In order to save the current state of RANLUX, we need an integer array of size 25. A call to the subroutine RLUXUT saves the necessary information in this array. We can save this array in order to read it at a later time and start the sequence of random numbers from the same point. The necessary code is:

```
integer,parameter          :: NSEEDS = 25
integer,dimension(NSEEDS)  :: seeds
!-------------------------------------------
open(unit=11,file='ranlux.seed')
call RLUXUT(seeds)
write(11,*)seeds
close(11)
```

In order to start RANLUX from a previously saved state we call the subroutine RLUXIN as follows:

```
integer,parameter          :: NSEEDS = 25
integer,dimension(NSEEDS)  :: seeds
open(unit=11,file='ranlux.seed')
read(11,*)seeds
call RLUXIN(seeds)
```

We can generate random numbers one by one and store them in a scalar variable

```
real(8)                    :: r
!-------------------------------------------
call ranlux(r,1)
```

```
 print *,r
```

or generate many random numbers with one call and store them in a one dimensional array

```
 integer,parameter          :: NR=20
 real(8),dimension(NR)      :: randoms
!-----------------------------------------
 call ranlux(randoms,NR)
 print *,randoms
```

where the parameter NR is set equal to the desired value. The program in the file test_ranlux.f90 implements all of the above tasks and we list it below:

```
program use_ranlux
 implicit none
 integer,parameter          :: NSEEDS = 25
 integer,dimension(NSEEDS)  :: seeds
 integer                    :: seed,ranlux_level
 integer(8)                 :: icount
 real(8)                    :: r
 integer,parameter          :: NR=20
 real(8),dimension(NR)      :: randoms
!-----------------------------------------
!start from a new seed:
 seed         = 58266273
 ranlux_level = 2
 call RLUXGO(ranlux_level,seed,0,0)
!-----------------------------------------
!generate random numbers one by one:
 do icount = 1,10
  call ranlux(r,1)
  print *,r
 enddo
!generate random numbers in an array:
 call ranlux(randoms,NR)
 print '(1000G28.17)',randoms
!-----------------------------------------
!save state of ranlux:
 open(unit=11,file='ranlux.seed')
 call RLUXUT(seeds)
 write(11,'(5I20)')seeds
 close(11)
!-----------------------------------------
!generate some randoms:
 call ranlux(randoms,NR)
```

```
 print '(A,1000G28.17)','#FIRST :',randoms
!-----------------------------------------
!read state of ranlux:
 open(unit=11,file='ranlux.seed')
 read(11,*)seeds
 call RLUXIN(seeds)
!-----------------------------------------
!generate same randoms!
 call ranlux(randoms,NR)
 print '(A,1000G28.17)','#SECOND:',randoms
end program use_ranlux
```

Compile the file together with the file `ranlux.F`, which contains the `RANLUX` code, and run it with the commands

```
> gfortran test_ranlux.f90 ranlux.F -o ranlux
> ./ranlux
```

## 11.3 Random Walks

Consider a particle which is located at one of the sites of a two dimensional square lattice. After equilibrating at this position, it can jump randomly to one of its nearest neighbor positions. There, it might need some time to equilibrate again before jumping to a new position. During this time, the momentum that it had at its arrival is lost, therefore the next jump is made without "memory" of the previous position where it came from. This process is repeated continuously. We are not interested in the mechanism that causes the jumping[11], and we seek a simple phenomenological description of the process.

Assume that the particle jumps in each direction with equal probability and that each jump occurs after the same time $\tau$. The minimum distance between the lattice sites is $a$ (lattice constant). The vector that describes the change of the position of the particle during the $i$-th jump is a random variable $\vec{\xi}_i$ and it always has the same magnitude $|\vec{\xi}_i| = a$. This means that, given the position $\vec{r}_k$ of the particle at time $t_k = k\tau$, its position $\vec{r}_{k+1}$ at time $t_{k+1} = (k+1)\tau = t_k + \tau$ is

$$\vec{r}_{k+1} = \vec{r}_k + \vec{\xi}_k \,, \tag{11.21}$$

[11] It could be e.g. thermally stimulated sound waves, the quantum tunneling effect etc.

where

$$\vec{\xi}_k = \begin{cases} a\hat{x} & \text{with probability } \frac{1}{4} \\ -a\hat{x} & \text{with probability } \frac{1}{4} \\ a\hat{y} & \text{with probability } \frac{1}{4} \\ -a\hat{y} & \text{with probability } \frac{1}{4} \end{cases} . \tag{11.22}$$

The vectors $\vec{\xi}_i$ and $\vec{\xi}_j$ are uncorrelated for $i \neq j$ and we have that

$$\langle \vec{\xi}_i \cdot \vec{\xi}_j \rangle = \langle \vec{\xi}_i \rangle \cdot \langle \vec{\xi}_j \rangle . \tag{11.23}$$

The possible values of $\vec{\xi}_i$ are equally probable, therefore we obtain

$$\langle \vec{\xi}_i \rangle = \vec{0} . \tag{11.24}$$

This is because the positive and negative terms in the sum performed in the calculation of $\langle \vec{\xi}_i \rangle$ occur with the same frequency and they cancel each other. Therefore $\langle \vec{\xi}_i \cdot \vec{\xi}_j \rangle = 0$ for $i \neq j$. Since the magnitude of the vectors $|\vec{\xi}_i| = a$ is constant, we obtain

$$\langle \vec{\xi}_i \cdot \vec{\xi}_j \rangle = a^2 \delta_{i,j} . \tag{11.25}$$

The probability for a path $C_N$ of length $N$ to occur is[12]

$$p(C_N) = \frac{1}{z^N} , \tag{11.26}$$

where $z = 4$ is the number of nearest neighbors of a lattice site. This probability depends on the length of the path and not on its geometry. This can be easily seen using the obvious relation $p(C_{N+1}) = \frac{1}{z} p(C_N)$, since there are exactly $z$ equally probable cases. The partition function is

$$Z_N = z^N , \tag{11.27}$$

and it counts the number of different paths of length $N$.

After time $t = N\tau$ the particle is displaced from its original position by

$$\vec{R} = \sum_{i=1}^{N} \vec{\xi}_i . \tag{11.28}$$

[12] I.e. after time $t = N\tau$, not the physical length of the path formed by the links that the particle has crossed. We also count the jumps to sites that the particle has already visited.

The average value of the displacement vanishes

$$\langle \vec{R} \rangle = \sum_{i=1}^{N} \langle \vec{\xi}_i \rangle = \vec{0}\,. \tag{11.29}$$

The expectation value of the displacement squared is non zero

$$\langle R^2 \rangle = \langle \vec{R} \cdot \vec{R} \rangle = \sum_{i,j=1}^{N} \langle \vec{\xi}_i \cdot \vec{\xi}_j \rangle = a^2 \sum_{i,j=1}^{N} \delta_{i,j} = a^2 N\,. \tag{11.30}$$

The conclusion is that the random walker has been displaced from its original position rather slowly

$$R_{rms} = \sqrt{\langle R^2 \rangle} = a\sqrt{N} \propto \sqrt{t}\,. \tag{11.31}$$

For a particle with a non zero average velocity we expect that $R_{rms} \propto t$.

Equation (11.31) defines the *critical exponent* $\nu$

$$\langle R^2 \rangle \sim N^{2\nu}\,, \tag{11.32}$$

where $\sim$ means asymptotic behavior in the limit $N \to \infty$. For a classical walker $\nu = 1$, whereas for the random walker $\nu = \frac{1}{2}$.

The Random Walker (RW) model has several variations, like the Non Reversal Random Walker (NRRW) and the Self Avoiding Walk (SAW) . The NRRW model is defined by excluding the vector pointing to the previous position of the walker and by selecting the remaining vectors $\vec{\xi}_i$ with equal probability. The SAW is a NRRW with the additional requirement that, when the walker ends in a previously visited position, the ... walking ends! Some models studied in the literature include, besides the infinite repulsive force, an attractive contribution to the total energy for every pair of points of the path that are nearest neighbors. In this case, each path is weighted with the corresponding Boltzmann weight according to equation (12.4).

For the NRRW, equation (11.32) is similar to that of the RW, i.e. $\nu = \frac{1}{2}$. Even though the paths differ microscopically, their long distance properties are the same. They are examples of models belonging to the same *universality class* according to the discussion in section 13.1.

This is not the case for the SAW. For this system we have that [47]

$$\langle R^2 \rangle^{\text{SAW}} \sim N^{2\nu} \qquad \nu = \frac{3}{4}\,, \tag{11.33}$$

therefore the typical paths in this model are longer than those of the RW. If we introduce a nearest neighbor attraction according to the previous discussion, then there is a critical temperature $\beta_c$ such that for $\beta < \beta_c$ we have similar behavior given by equation (11.33), whereas for $\beta > \beta_c$ the attractive interaction dominates, the paths collapse and we obtain $\nu = 1/3 < \nu_{RW}$. For $\beta = \beta_c$ we have that $\nu = \frac{1}{2}$. For more information we refer the reader to the book of Binder and Heermann [7].

In order to write a program that simulates the RW we apply the following algorithm:

1. Set the number of the random walks to be generated
2. Set the number of steps of each walk
3. Set the initial position of the walk
4. At each step on the walk, pick a random direction with equal probability
5. After the walk is completed, measure $\vec{R}$, $R^2$, etc
6. After all walks have been generated, compute the expectation values of the measured quantities and the statistical error of their measurement.

All we need to explain is how to program the choice of "random direction". The program is in the file `rw.f90`

```
program random_walker
 implicit none
 integer,parameter :: Nwalk = 1000
 integer,parameter :: Nstep = 100000
 integer           :: iwalk,istep,ir
 real(8)           :: x,y
 real(8)           :: drandom
 integer           :: seed
 common /randoms/     seed

 seed = 374676287
 open(unit=20,file='dataR')
 do iwalk = 1,Nwalk
  x = 0.0D0 ; y = 0.0D0
  open(unit=21,file='data')
  do istep=1,Nstep
   ir = INT(drandom()*4)
   select case(ir)
```

```
    case(0)
     x = x + 1.0D0
    case(1)
     x = x - 1.0D0
    case(2)
     y = y + 1.0D0
    case(3)
     y = y - 1.0D0
   end select
   write(21,*) x,y
  enddo !do istep=1,Nstep
  close(21)
  call sleep(2)
  write(20,*) x*x+y*y
  call flush(20)
 enddo !do iwalk = 1,Nwalk
end program random_walker
```

The length of the paths is `Nstep` and the number of the generated paths is `Nwalk`. Their values are hard coded and a run using different values requires recompilation. The results are written to the files `dataR` and `data`. The square of the final displacement of the walker $R^2$ is written to `dataR` and the coordinates $(x, y)$ of the points visited by the walker in each path is written to `data`. In order to make the contents of the files available immediately, we empty the I/O buffers by a call to the subroutine `flush(unit)`. The file `data` is truncated at the beginning of each path, therefore it contains the coordinates of the current path only.

Each path is made of `Nstep` steps. The random vector $\vec{\xi}_{\texttt{istep}}$ is chosen and it is added in the current position $\vec{r}_{\texttt{istep}} = (x, y)$. The choice on $\vec{\xi}_{\texttt{istep}}$ is made in the line

```
   ir = INT(drandom()*4)
```

where the variable `ir` $= 0, 1, 2, 3$ because the function `INT` returns the integer part of a `real`. The values of `ir` correspond to the four possible directions of $\vec{\xi}$. We use the construct `select case(ir)` in order to move in the direction chosen by `ir`. Depending on its value, the control of the program is transferred to the command that moves the walker to the corresponding direction.

Compiling and running the program can be done with the commands

```
> gfortran rw.f90 drandom.f90 -o rw
> ./rw
```

Because of the command `call sleep(2)`, the program temporarily halts execution for 2 seconds at the end of each generated path (you should remove this line at the production stage). This allows us to monitor the generated paths graphically. During the execution of the program, use `gnuplot` in order to plot the random walk which is currently stored in the file `data`:

```
gnuplot> plot "data" with lines
```

Repeat for as many times as you wish to see new random walks. The automation of this process is taken care in the script `eternal-rw`:

```
> ./rw &
> ./eternal-rw &
> killall rw eternal-rw gnuplot
```

The last command ends the execution of all programs.

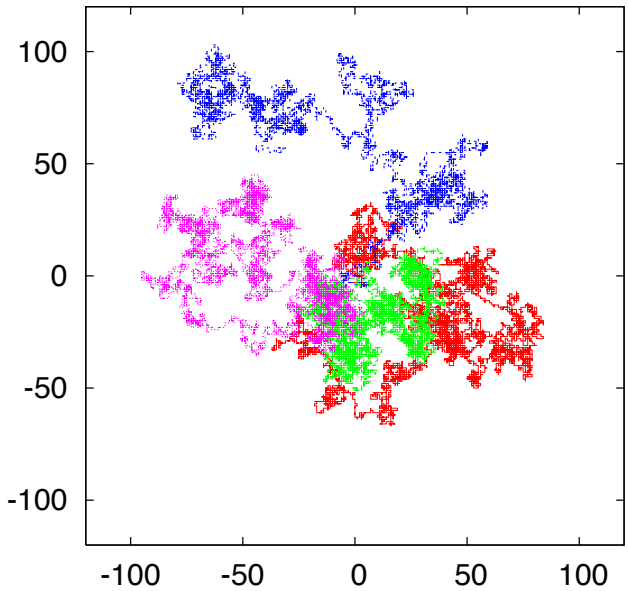



Figure 11.8: Four typical paths of the RW for $N = 10000$.

Some typical paths are shown in figure 11.8. Figure 11.9 shows the results for the expectation value $\langle R^2 \rangle$ for $N = 10, \ldots, 100000$ which confirm equation (11.30) $\langle R^2 \rangle = N$. You can reproduce this figure as follows:

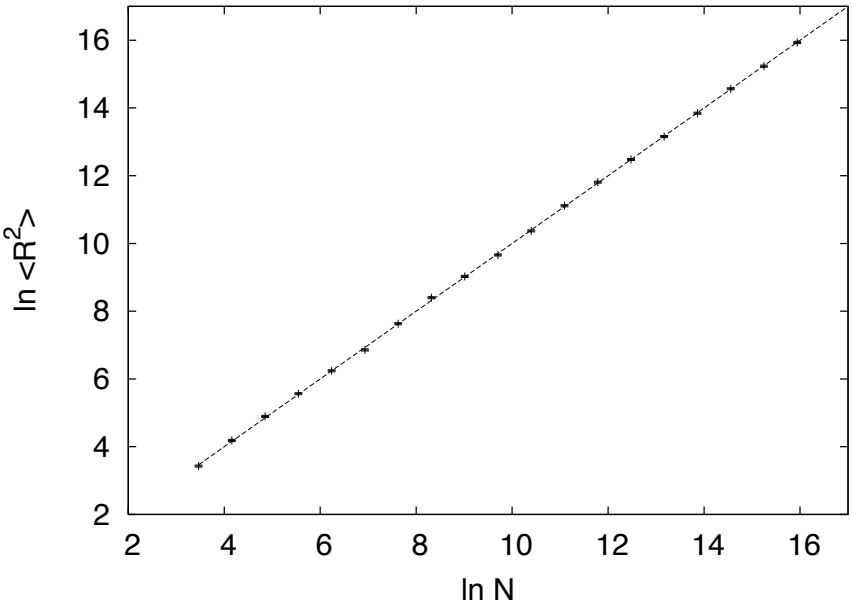



Figure 11.9: Numerical confirmation of the relation $\langle R^2 \rangle = N$ for $N = 10, \ldots, 100000$. The straight line is the fit of the data to the function $y = ax$ with $a = 0.9994(13)$.

1. Set the values of `Nwalk` and `Nstep` in the file `rw.f90`. Delete the commands `call sleep(2)` and `write(21,*) x,y` and compile the code.

2. Run the program and analyze the data in the file `dataR`:

```
> ./rw
> awk '{av += $1}END{print av/NR}' dataR
```

   Write the results in a file `r2.dat` in two columns with the length of the paths $N$ in the first column and with $\langle R^2 \rangle$ in the second. The command[13] `{av+=$1}` in the `awk` program adds the first column of each line of the file `dataR` to the variable `av`. After reading the whole file, the command `END{print av/NR}`, prints the variable `av` divided by the number of lines in the file (`NR` = "Number of Records"). This is a simple way for computing the mean of the first column of the file `dataR`.

[13]The command `av+=$1` is equivalent to `av=av+$1`.

3. Use a linear squares method in order to find the optimal line $y = ax + b$ going through the points $(\ln N, \ln\langle R^2\rangle)$. You can also use the `fit` command in `gnuplot` as follows:

```
gnuplot> fit  a*x+b "r2.dat" u (log($1)):(log($2)) via a,b
```

4. Construct the plot with the command:

```
gnuplot> plot a*x+b,"r2.dat" u (log($1)):(log($2)) w e
```

The obtained results are meaningless without their statistical errors. Since each measurement is statistically independent, the true expectation value is approached in the limit of infinite measurements with a speed proportional to $\sim 1/\sqrt{M}$, where $M$ is the number of measurements. For the same reason[14], the statistical error is given by equation (11.3), e.g.

$$\delta\langle R^2\rangle = \sqrt{\frac{1}{M-1}\left(\frac{1}{M}\sum_{i=1}^{M}(R_i^2)^2 - \left(\frac{1}{M}\sum_{i=1}^{M}R_i^2\right)^2\right)}\,. \tag{11.34}$$

We can add the calculation of the error in the program in `rw.f90` or we can leave this task to external utilities. For example we can use the `awk` script, which is written in the file `average`:

```
#!/usr/bin/awk -f
{
  av += $1;      # the sum of data
  er += $1*$1;   # the sum of squares of data
}
END{
  av /= NR;      # NR = "Number of Records" = number of lines
  er /= NR;
  # formula for error of uncorrelated measurements
  er  = sqrt( (er - av*av)/(NR-1) );
  print av, "+/-", er;
}
```

The contents of this file is an example of a script *interpreted* by the `awk` program. The operating system knows which program to use for the

[14] If there exist statistical correlations between measurements, they should be taken into consideration. This will be discussed in detail in the following chapters.

interpretation by reading the first line `#!/bin/awk -f` where the first two characters of the program should be exactly `#!`. For the commands to be interpreted and executed, one has to make the script executable using the command `chmod a+x average`. Then the command

```
> ./average dataR
```

executes the script using the `awk` interpreter. We remind to the reader that the commands between curly brackets `{ ... }` are executed by `awk` for every line of the file `dataR`. The commands between `END{ ... }` are executed after the last line of the file has been read[15]. Therefore the lines

```
av += $1;       #  the sum of data
er += $1*$1;    # the sum of squares of data
```

add the first column of the file `dataR` and its square to the variables `av` and `er` respectively. The commands

```
av /= NR;       # NR = "Number of Records" = number of lines
er /= NR;
```

are executed after the whole file `dataR` has been read and divide the variables `av` and `er` with the predefined variable `NR` which counts the total number of lines read so far. The last lines of the script compute the error according to equation (11.34) and print the final result. The shell script in the file `rw1-anal.csh` codes all of the above commands in a script. Read the comments in the file for usage instructions.

## 11.4 Problems

1. Reproduce the results shown in figure 11.6 and confirm the validity of equation (11.5).

2. Generate a sequence of pseudorandom numbers which follow a Gaussian distribution with standard deviation $\sigma = 1/\sqrt{2}$. Construct the plot of relative frequencies together with the plot of the probability density function.

[15] You can also execute a set of commands *before* the file is read by putting them between `BEGIN{ ... }`

3. Generate a sequence of pseudorandom numbers which follow the Cauchy distribution with $c = 1$. Construct the plot of relative frequencies together with the plot of the probability density function.

4. Write a program that calculates the period of the function `drandom()`. Check whether the numbers 0 and 1 belong to the sequence.

5. Compute the CPU time cost of the random number generation as follows: If you have an executable file, e.g. `random`, run the `/usr/bin/time` command with `./random` as its argument:

```
> /usr/bin/time ./random
```

   Upon exit of the command, the program `/usr/bin/time` prints the total CPU time in seconds to the `stderr`. Compute the time needed to generate $10^9$ random numbers using the function `drandom()` and the subroutines `random_number` and `ranlux`. For `RANLUX`, measure the CPU time for each value of `ranlux_level` from 1 to 4. How does this time depend on whether the random numbers are generated one by one or in groups of 1000 by calls to `random_number` and `ranlux` having as an argument an array of size 1000? How does the time change if the random numbers are generated in groups of 10000 instead? (Hint: see the file `performance_ran.f90`)

6. For each of the random number generators `drandom()`, `random_number` and `ranlux`, generate 10 random numbers. Then save the state of the generator in a file. Then generate 10 more random numbers. Read from the file the saved state of the generator and generate 10 random numbers. Check that the last two sequences of random numbers are identical.

7. Make the appropriate changes in the file `seed.f90` so that it can be used for seeding `RANLUX`. Do it in two different ways: (a) by generating one seed and use `RLUXGO` for the initialization. (b) By generating 25 seeds and use `RLUXIN` for the initialization.

8. Show that if the expectation values of the vectors $\langle\vec{\xi}_i\rangle = \vec{v}\tau$ then $\langle\vec{R}\rangle = \vec{v}\tau N$ and we obtain a linear relation between displacement–length of path. The quantity $v$ is the expectation value of the speed of the particle. Compute $\langle R^2\rangle$ for large values of $N$.

9. Confirm the relations computed in the previous problem numerically. In your program, set the first line in (11.22) equal to $1/2$

and the rest equal to $1/6$. Compute the expectation values $\langle(\xi_i)_x\rangle$ and $\langle(\xi_i)_y\rangle$ and use them to calculate the average speed of the particle. Check the validity of the relations $\langle R^2\rangle \sim N^{\alpha}$ and $\langle R_x\rangle \sim N^{2a_x}$ $\langle R_y\rangle \sim N^{2a_y}$. What is the relation between $a$, $a_x$ and $a_y$?

10. Make the appropriate changes in the file `rw.f90` so that the user can enter the values `Nwalk` and `Nstep` interactively using the command line arguments. For example, if she wants to generate 100 random walks with $N = 2000$, she should run the command `./rw 100 2000`. You will need to use the Fortran intrinsic functions `GETARG` και `IARGC`. (Hint: Look in the file `rw1.f90`)

11. We know that for the RW we have that $\langle\vec{R}\rangle = \vec{0}$. Calculate $\langle x\rangle$ and $\langle y\rangle$ numerically for $N = 100, 100000$. Are they really equal to zero? Why? How does this depend on the number of measurements?

12. Compute the expectation value of the number of returns of the RW to his initial position as a function of $N$. What happens as $N \to \infty$? Why?

13. Reproduce figure 11.9 for the RW.

14. Write a program that implements the NRRW and reproduce the results in figure 11.9 for the NRRW.

15. In the program `rw.f90` the RW's position is determined by two `REAL(8)` variables `x,y`. The next position is calculated by the statements `x=x+1.0D0, y=y+1.0D0`. What are the limitations on the size of random walks that can be studied with this choice? What happens if one uses `REAL` variables `x,y` instead? Take into account the fact that $\langle R^2\rangle = N$.

16. Repeat the previous problem by using `INTEGER(8)` variables `x,y`. The next position is calculated by the statements `x=x+1, y=y+1`. Discuss the pros and the cons of each choice.

17. Repeat the previous problems by using `INTEGER(4)` variables `x,y`. Discuss the pros and the cons of each choice by considering also the running time of the program. Use the command `/usr/bin/time`.

18. Write a straightforward code that implements the SAW. How big $N$ can you simulate? Check whether the CPU time for computing a given number of random walks increases exponentially with $N$.

Search the internet for the most efficient algorithm that simulates the SAW for large $N$.

# Chapter 12

# Monte Carlo Simulations

In this chapter we review the basic principles of Monte Carlo simulations in statistical mechanics. In the introduction, we review some of the fundamental concepts of statistical physics. The reader should have a basic understanding of concepts like the canonical ensemble, the partition function, the entropy, the density of states and the quantitative description of fluctuations of thermodynamic quantities. For a more in depth discussion of these concepts, see [4, 40, 48, 49, 50, 51].

For most of the interesting systems, the partition function cannot be calculated analytically, and in such a case we may resort to a numerical computation. This is what is done most effectively using Monte Carlo simulations, which consist of collecting a statistical sample of states of the system with an appropriately chosen probability distribution. It is remarkable that, by collecting a sample which is a tiny fraction of the total number of states, we can perform an accurate calculation of its thermodynamic quantities[1]. But this is no surprise: it happens in our labs all the time[2]!

[1]For example, for the $d = 2$, $L = 100$ Ising model, we have $2^{100\times 100} = 2^{10000} \approx 10^{3010}$ states. A typical sample yielding a very accurate measurement consists of $\approx 10^7$ states, i.e. a fraction of $\approx 10^{-3003}$! This fraction becomes many orders of magnitude smaller for realistic complex systems studied in today's supercomputers.

[2]For a gas formed by $10^{22}$ molecules which has volume equal to 1 lt in room temperature and atmospheric pressure, the average velocity of its molecules is $\approx 100ms^{-1}$. This means that the typical de Broglie wavelength of the molecules is $\lambda \approx 10^{-10}m$. If we estimate that the volume occupied by each molecule is of order $\lambda^3$, then the number of states that each molecule can be is $\approx 10^{27}$. Therefore the system can be in $\approx (10^{27})^{10^{22}}$ different states. If we assume that on the average the molecules collide $10^9$ times per second, then we have $\approx 10^{31}$ changes of states per second. In order that the system visits all possible states, the time needed is $10^{10^{23}}$ times the age of the universe [4].

## 12.1 Statistical Physics

Statistical physics describes systems with a very large number of degrees of freedom $N$. For simple macroscopic systems $N \approx 10^{23} - 10^{44}$. For such systems, it is practically impossible to solve the microscopic equations that govern their dynamics. Even if we could, the solution would have had much more information than we need (and capable of analyzing!). It is enough, however, to know a small number of bulk properties of the system in order to have a useful description of it. E.g. it is enough to know the internal energy and magnetization of a magnet or the energy and density of a fluid instead of the detailed knowledge of the position, momentum, energy and angular momentum of each particle they are made of. These quantities provide a thermodynamic description of a system. Statistical physics makes an attempt to derive these quantities from the microscopic degrees of freedom and their dynamics given by the Hamiltonian of the system.

Consider a system which can be in a set of *discrete* states which belong to a countable set $\{\mu\}$. The energy spectrum of those states is assumed to consist of discrete values[3] $E_0 < E_1 < \ldots < E_n < \ldots$. This system is in contact and interacts with a large heat reservoir which has temperature $\beta = 1/kT$. The contact with the reservoir results in *random* transitions which change the energy of the system[4]. The system is described by the weights $w_\mu(t)$ which give the probability to find the system in a state $\mu$ at time $t$. These weights are the connection between the microscopic and statistical description of the system. When this system is in thermal equilibrium with the reservoir, its statistical properties are described by the, so called, *canonical ensemble*.

Let $R(\mu \to \nu)$ be the transition rates from the state $\mu \to \nu$, i.e.

$$R(\mu \to \nu)dt = \text{Transition probability } \mu \to \nu \text{ in time } dt\,, \tag{12.1}$$

which depend on the interaction between the system and the thermal reservoir. The *master equation* for the weights $w_\mu(t)$ is

$$\begin{aligned} \frac{dw_\mu(t)}{dt} &= \sum_\nu \{w_\nu(t)R(\nu \to \mu) - w_\mu(t)R(\mu \to \nu)\} \\ \sum_\mu w_\mu(t) &= 1\,. \end{aligned} \tag{12.2}$$

[3] $E_0$ is the *ground state* energy of the system.

[4] An isolated system always has constant energy. Such a system is studied in the microcanonical ensemble.

The first of the above equations tells us that the change in $w_\mu(t)$ is equal to the rate that the system comes into the state $\mu$ from *any* other state $\nu$, minus the rate of leaving the state $\mu$. The second equation is a result of the probability interpretation of the weights $w_\mu(t)$ and states that the probability of finding the system in any state is equal to 1 at all times.

The transition rates $R(\mu \to \nu)$ are assumed to be time independent and then the above system of equations for $w_\mu(t)$ is linear with real parameters. This, together with the constraint $0 \leq w_\mu(t) \leq 1$, implies that[5], in the large time limit, $\frac{dw_\mu(t)}{dt} = 0$ and the system reaches equilibrium. Then, the $w_\mu(t)$ converge to finite numbers $p_\mu \geq 0$. These are the equilibrium occupation probabilities

$$p_\mu = \lim_{t\to\infty} w_\mu(t)\,, \qquad \sum_\mu p_\mu = 1\,. \tag{12.3}$$

For a system in thermodynamic equilibrium with a reservoir in temperature $T$, with $\beta = 1/kT$, the probabilities $p_\mu$ follow the Boltzmann distribution (Gibbs 1902)

$$p_\mu = \frac{1}{Z}\,\mathrm{e}^{-\beta E_\mu}\,, \tag{12.4}$$

and define the, so called, *canonical ensemble*. The parameter $\beta$ will be frequently referred to as simply "the temperature" of the system, although, strictly speaking, it is the inverse of it. Its appearance in the exponential in equation (12.4), defines a characteristic energy scale of the system. The Boltzmann constant $k \approx 1.38 \times 10^{-23} JK^{-1}$ is simply a conversion constant between units of energy[6].

The normalization $Z$ in equation (12.4) is the so called partition function of the system. The condition $\sum_\mu p_\mu = 1$ implies

$$Z(\beta) = \sum_\mu \mathrm{e}^{-\beta E_\mu} \tag{12.5}$$

The measurement of a physical quantity, or *observable*, of a thermodynamic system has a stochastic character. For systems with very large number of degrees of freedom $N$, one is interested only in the average

[5]Note that equation (12.2) can be written in the form $\frac{dw_\mu(t)}{dt} = \sum_\nu \mathcal{R}_{\mu\nu} w_\nu(t)$, where the matrix $\mathcal{R}_{\mu\nu}$ has real, constant elements.

[6]It is *not* a fundamental constant of nature like $c$, $\hbar$, $G$, .... Temperature *is* an energy scale and the fact that it is customary to measure it in degrees Kelvin or other, is a historical accident due to the ignorance of the microscopic origin of heat exchange at the times of the original formulation of thermodynamics.

value of such a quantity. This is because the probability of measuring the quantity to take a value significantly different from its average is ridiculously small. The average, or *expectation value*, $\langle \mathcal{O} \rangle$ of a physical observable $\mathcal{O}$ whose value in a state $\mu$ is $\mathcal{O}_\mu$ is equal to

$$\langle \mathcal{O} \rangle = \sum_\mu p_\mu \mathcal{O}_\mu = \frac{1}{Z} \sum \mathcal{O}_\mu \mathrm{e}^{-\beta E_\mu} \,. \tag{12.6}$$

As we will see later, the standard deviation $\Delta\mathcal{O}$ for a typical thermodynamic system is such that

$$\frac{\Delta \mathcal{O}}{\mathcal{O}} \sim \frac{1}{\sqrt{N}} \,, \tag{12.7}$$

which is quite small for macroscopic systems[7]. In such cases, the fluctuations of the values of $\mathcal{O}$ from its expectation value $\langle \mathcal{O} \rangle$ can be neglected. The limit $N \to \infty$ is the so called *thermodynamic limit*, and it is in this limit in which we are studying systems in statistical mechanics. Most systems in the lab are practically in this limit, but in the systems simulated on a computer we may be far from it. The state of the art is to invent methods which can be used to extrapolate the results from the study of the finite system to the thermodynamic limit efficiently.

Because of (12.5), the partition function encodes all the statistical information about the system. It is not just a simple function of one or more variables, but it counts all the states of the system with the correct weight. Its knowledge is equivalent to being able to compute any thermodynamic quantity like, for example, the expectation value of the energy $\langle E \rangle$ of the system[8]:

$$\begin{aligned} U \;\equiv\; \langle E \rangle &= \frac{1}{Z} \sum_\mu E_\mu \, \mathrm{e}^{-\beta E_\mu} = -\frac{1}{Z} \sum_\mu \frac{\partial}{\partial \beta} \mathrm{e}^{-\beta E_\mu} = -\frac{1}{Z} \frac{\partial}{\partial \beta} \sum_\mu \mathrm{e}^{-\beta E_\mu} \\ &= -\frac{1}{Z} \frac{\partial Z}{\partial \beta} = -\frac{\partial \ln Z}{\partial \beta} \,. \end{aligned} \tag{12.8}$$

Similarly, one can calculate the specific heat from

$$C = \frac{\partial U}{\partial T} = \frac{\partial \beta}{\partial T} \frac{\partial U}{\partial \beta} = (-k\beta^2)(-\frac{\partial^2 \ln Z}{\partial \beta^2}) = k\beta^2 \frac{\partial^2 \ln Z}{\partial \beta^2} \,. \tag{12.9}$$

[7]E.g. for $N \sim 10^{23}$ we have that $\Delta\mathcal{O}/\mathcal{O} \sim 10^{-11}$ and the measurements of $\mathcal{O}$ fluctuate at the 11th significant digit of their value. This is usually much smaller than other experimental errors.

[8]In thermodynamics, $\langle E \rangle$ corresponds to the internal energy $U$ of the system.

## 12.2 Entropy

The entropy $S$ of a thermodynamic system is defined by

$$S = -\frac{\partial F}{\partial T}\,, \quad F = U - TS\,, \tag{12.10}$$

where $F$ is the *free energy* if the system. We will attempt to provide microscopic definitions that are consistent with the above equations.

We define the free energy from the relation

$$\mathrm{e}^{-\beta F} = Z \equiv \sum_\mu \mathrm{e}^{-\beta E_\mu}\,, \tag{12.11}$$

or equivalently

$$F = -\frac{1}{\beta}\ln Z\,. \tag{12.12}$$

Note that for $T \to 0$ the free energy becomes the ground state energy[9]. Indeed, as $\beta \to \infty$ only the lowest energy term in equation (12.11) survives. For this reason, equation (12.10) gives $\lim_{T\to 0} S = 0$, which is the third law of thermodynamics.

The definition (12.11) is consistent with (12.10) since

$$U = -\frac{\partial \ln Z}{\partial \beta} = -\frac{\partial}{\partial \beta}(-\beta F) = F + \beta\frac{\partial F}{\partial \beta} = F - T\frac{\partial F}{\partial T} = F + TS\,. \tag{12.13}$$

The relation of the entropy $S$ to the microscopic degrees of freedom can be derived from equations (12.11) and (12.10):

$$\frac{S}{k} = \frac{U-F}{kT} = \beta(U-F) = \beta(\sum_\mu p_\mu E_\mu + \frac{1}{\beta}\ln Z)\,. \tag{12.14}$$

But

$$p_\mu = \frac{\mathrm{e}^{-\beta E_\mu}}{Z} \Rightarrow E_\mu = -\frac{1}{\beta}(\ln p_\mu + \ln Z)\,, \tag{12.15}$$

therefore

$$\begin{aligned}\frac{S}{k} &= \beta\sum_\mu\left(-\frac{1}{\beta}(\ln p_\mu + \ln Z)p_\mu + \frac{1}{\beta}\ln Z\right)\\ &= -\sum_\mu p_\mu \ln p_\mu - \ln Z\sum_\mu p_\mu + \ln Z\\ &= -\sum_\mu p_\mu \ln p_\mu\,.\end{aligned} \tag{12.16}$$

[9]For strict equality it is necessary that the ground state is not degenerate as it happens in the case of spontaneous symmetry breaking.

Finally

$$S = -k \sum_{\mu} p_\mu \ln p_\mu \,. \tag{12.17}$$

Let's analyze the above relation in some special cases. Consider a system[10] where all possible states have the same energy. For such a system, using equation (12.17), we obtain that

$$p_\mu = \frac{1}{g} = \text{const.} \Rightarrow S = k \ln g \,. \tag{12.18}$$

Therefore, the entropy simply counts the number of states of the system. This is also the case in the microcanonical ensemble. Indeed, equation (12.18) is also valid for the distribution

$$p_\mu = \begin{cases} \frac{1}{g(E)} & E_\mu = E \\ 0 & E_\mu \neq E \end{cases} \,, \tag{12.19}$$

which can be considered to be equivalent to the microcanonical ensemble since it enforces $E_\mu = E =$ const. Equation (12.19) can viewed as an approximation to a distribution sharply peaked at $E$. In such a case, $S$ counts, more or less, the number of states of the system with energy close to $E$.

In general, the function[11] $g(E)$ is defined to be equal to the number of states with energy equal to $E$. Then the probability $p(E)$ to measure energy $E$ in the canonical ensemble is

$$p(E) = \langle \delta_{E,E_\mu} \rangle = \sum_{\mu} p_\mu \delta_{E,E_\mu} = \frac{1}{Z} \sum \mathrm{e}^{-\beta E_\mu} \delta_{E,E_\mu} = \frac{1}{Z} \mathrm{e}^{-\beta E} \sum \delta_{E,E_\mu} \,. \tag{12.20}$$

Since $\sum_\mu \delta_{E,E_\mu} = g(E)$, we obtain

$$p(E) = \langle \delta_{E,E_\mu} \rangle = \frac{g(E)\, \mathrm{e}^{-\beta E}}{Z} \,. \tag{12.21}$$

For a generic system we have that

$$g(E) \sim E^{\alpha N} \,, \tag{12.22}$$

where $N$ is the number of degrees of freedom of the system and $\alpha$ is a constant. The qualitative behavior of the distribution (12.21) is shown in

[10] E.g. the random walker, two dimensional quantum gravity without matter.

[11] The notation $\Omega(E)$ is also frequently used and it is referred to as the *density of states*.

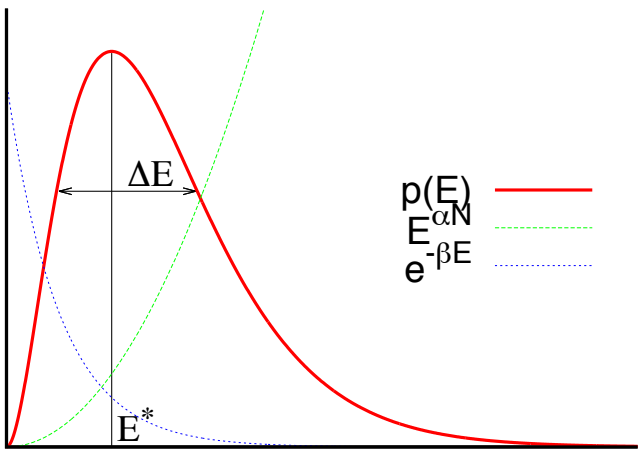



Figure 12.1: The probability $p(E)$ as a result of the competition of the Boltzmann factor $\mathrm{e}^{-\beta E}$ and the density of states $g(E) \sim E^{\alpha N}$. $E^*$ is the most probable value of the energy and $\Delta E$ is a measure of the energy fluctuations.

figure 12.1. For such a system the most probable values of the energy are sharply peaked around a value $E^*$ and the deviation $\Delta E$ is a measure of the energy fluctuations. The ratio $\Delta E/E$ drops with $N$ as $1/\sqrt{N}$. Indeed, the function[12]

$$\tilde{p}(E) = E^{\alpha N}\mathrm{e}^{-\beta E} = \mathrm{e}^{-\beta E - \alpha N \ln E} \tag{12.23}$$

has a maximum when

$$\left.\frac{\partial \ln \tilde{p}(E)}{\partial E}\right|_{E=E^*} = 0 \Rightarrow \left.\frac{\partial}{\partial E}(-\beta E + \alpha N \ln E)\right|_{E=E^*} = -\beta + \frac{\alpha N}{E^*} = 0 \tag{12.24}$$

or

$$E^* = \frac{\alpha}{\beta} N\,. \tag{12.25}$$

As the temperature increases ($\beta$ decreases), $E^*$ shifts to larger values. $E^*$ is proportional to the system size. By Taylor expanding around $E^*$ we

[12] $\tilde{p}(E)$ is proportional to $p(E)$ for fixed $\beta$. It is only defined for convenience.

obtain

$$\begin{aligned}\ln \tilde{p}(E) &= \ln \tilde{p}(E^*) + (E - E^*) \left.\frac{\partial \ln \tilde{p}(E)}{\partial E}\right|_{E=E^*} \\ &\quad + \frac{1}{2}(E - E^*)^2 \left.\frac{\partial^2 \ln \tilde{p}(E)}{\partial E^2}\right|_{E=E^*} + \dots \\ &= \ln \tilde{p}(E^*) + \frac{1}{2}(E - E^*)^2 \left(-\frac{\alpha N}{(E^*)^2}\right) + \dots , \end{aligned} \tag{12.26}$$

where we used equation (12.24) and computed $\left.\frac{\partial^2 \ln \tilde{p}(E)}{\partial E^2}\right|_{E=E^*}$. Therefore

$$p(E) \approx p(E^*) \mathrm{e}^{-\alpha N \frac{(E-E^*)^2}{2(E^*)^2}} , \tag{12.27}$$

which is a Gaussian distribution with standard deviation

$$\Delta E \sim \sqrt{\frac{(E^*)^2}{\alpha N}} = \sqrt{\frac{(\frac{\alpha N}{\beta})^2}{\alpha N}} \sim \frac{\sqrt{N}}{\beta} . \tag{12.28}$$

Therefore we confirm the relation (12.7)

$$\frac{\Delta E}{E^*} \sim \frac{\frac{\sqrt{N}}{\beta}}{\frac{N}{\beta}} = \frac{1}{\sqrt{N}} . \tag{12.29}$$

In the analysis above we assumed analyticity (Taylor expansion, equation (12.26)), which is not valid at a critical point of a phase transition in the thermodynamic limit.

Another important case where the above analysis becomes slightly more complicated is when the distribution $p(E)$ has more than one equally probable maxima[13] separated by a large probability barrier as shown in figure 12.2 like when the system undergoes a first order phase transition. Such a transition occurs when ice turns into water or when a ferromagnet looses its permanent magnetization due to temperature increase past its Curie point. In such a case the two states, ice – water / ferromagnet – paramagnet, are equally probable and coexist. This is qualitatively depicted in figure 12.2.

## 12.3 Fluctuations

The stochastic behavior of every observable $\mathcal{O}$ is given by a distribution function $p(\mathcal{O})$ which can be derived from the Boltzmann distribution

[13] When there are many local maxima, the absolute maximum dominates in the thermodynamic limit $N \to \infty$.

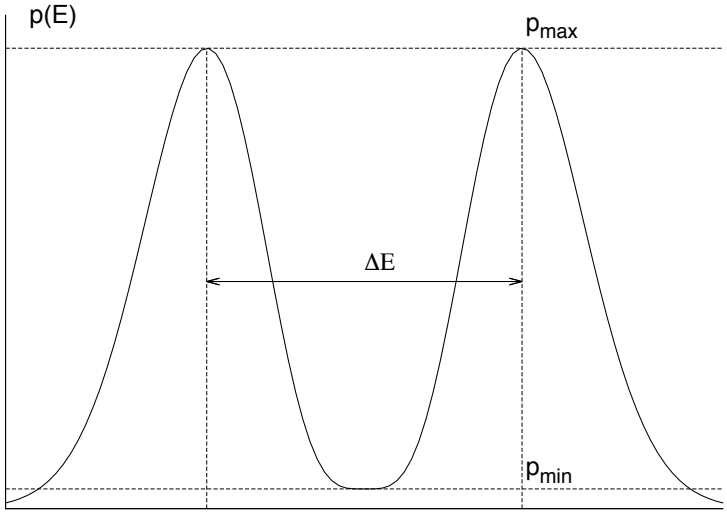



Figure 12.2: Two peak structure in the distribution $p(E)$ of the energy $E$ for a system undergoing a first order phase transition. The two maxima correspond to two coexisting states ("ice"–"water") and $\Delta E/N$ is the latent heat. In the thermodynamic limit $N \to \infty$, $R = p_{min}/p_{max}$ decreases like $R \sim \mathrm{e}^{-fA}$, where $A$ is the minimal surface separating the two phases and $f$ is the interface tension.

(12.4). Such a distribution is completely determined by its expectation value $\langle \mathcal{O} \rangle$ and all its higher order moments, i.e. the expectation values $\langle (\mathcal{O} - \langle \mathcal{O} \rangle)^n \rangle$, $n = 1, 2, 3. \ldots$. The most commonly studied moment is the second moment ($n = 2$)

$$(\Delta \mathcal{O})^2 \equiv \langle (\mathcal{O} - \langle \mathcal{O} \rangle)^2 \rangle = \langle \mathcal{O}^2 \rangle - \langle \mathcal{O} \rangle^2 \,. \tag{12.30}$$

For a distribution with a single maximum, $\Delta \mathcal{O}$ is a measure of the fluctuations of $\mathcal{O}$ away from its expectation value $\langle \mathcal{O} \rangle$. When $\mathcal{O} = E$ we obtain

$$(\Delta E)^2 \equiv \langle (E - \langle E \rangle)^2 \rangle = \langle E^2 \rangle - \langle E \rangle^2 \,, \tag{12.31}$$

and using the relations

$$\langle E^2 \rangle = \frac{1}{Z} \sum_\mu E_\mu^2 \mathrm{e}^{-\beta E_\mu} = \frac{1}{Z} \frac{\partial^2}{\partial \beta^2} \sum_\mu \mathrm{e}^{-\beta E_\mu} = \frac{1}{Z} \frac{\partial^2 Z}{\partial \beta^2} \,, \tag{12.32}$$

and

$$\langle E \rangle = \frac{1}{Z} \sum_\mu E_\mu \mathrm{e}^{-\beta E_\mu} = -\frac{1}{Z} \frac{\partial}{\partial \beta} \sum_\mu \mathrm{e}^{-\beta E_\mu} = -\frac{1}{Z} \frac{\partial Z}{\partial \beta} \,, \tag{12.33}$$

we obtain that

$$(\Delta E)^2 = \langle E^2 \rangle - \langle E \rangle^2 = \frac{1}{Z}\frac{\partial^2 Z}{\partial \beta^2} - \left(-\frac{1}{Z}\frac{\partial Z}{\partial \beta}\right)^2 = \frac{\partial^2 \ln Z}{\partial \beta^2}\,, \tag{12.34}$$

which, according to (12.9), is the specific heat

$$C = \frac{\partial \langle E \rangle}{\partial T} = k\beta^2 (\Delta E)^2\,. \tag{12.35}$$

This way we relate the specific heat of a system (a thermodynamic quantity) with the microscopic fluctuations of the energy.

This is true for every physical quantity which is linearly coupled to an external field (in the case of $E$, this role is played by $\beta$). For a magnetic system in a constant magnetic field $B$, such a quantity is the magnetization $M$. If $M_\mu$ is the magnetization of the system in the sate $\mu$ and we assume that its direction is parallel to the direction of the magnetic field $\vec{B}$, then the Hamiltonian of the system is

$$H = E - BM\,, \tag{12.36}$$

and the partition function is

$$Z = \sum_\mu \mathrm{e}^{-\beta E_\mu + \beta B M_\mu}\,. \tag{12.37}$$

"Linear coupling" signifies the presence of the linear term $BM$ in the Hamiltonian. The quantities $B$ and $M$ are called conjugate to each other. Other well known conjugate quantities are the pressure/volume ($P/V$) in a gas or the chemical potential/number of particles ($\mu/N$) in the grand canonical ensemble.

Because of this linear coupling we obtain

$$\langle M \rangle = \frac{1}{Z}\sum_\mu M_\mu\, \mathrm{e}^{-\beta E_\mu + \beta B M_\mu} = \frac{1}{\beta Z}\frac{\partial Z}{\partial B} = -\frac{\partial F}{\partial B}\,, \tag{12.38}$$

which is analogous to (12.8). The equation corresponding to (12.34) is obtained from (12.30) for $\mathcal{O} = M$

$$(\Delta M)^2 \equiv \langle (M - \langle M \rangle)^2 \rangle = \langle M^2 \rangle - \langle M \rangle^2\,. \tag{12.39}$$

From (12.37) we obtain

$$\langle M^2 \rangle = \frac{1}{Z}\sum_\mu M_\mu^2\, \mathrm{e}^{-\beta E_\mu + \beta B M_\mu} = \frac{1}{\beta^2 Z}\frac{\partial^2 Z}{\partial B^2}\,, \tag{12.40}$$

therefore

$$(\Delta M)^2 = \frac{1}{\beta^2}\left\{\frac{1}{Z}\frac{\partial^2 Z}{\partial B^2} - \frac{1}{Z^2}\left(\frac{\partial Z}{\partial B}\right)^2\right\} = \frac{1}{\beta^2}\frac{\partial^2 \ln Z}{\partial B^2} = \frac{1}{\beta}\frac{\partial\langle M\rangle}{\partial B}\,. \quad (12.41)$$

The magnetic susceptibility $\chi$ is defined by the equation

$$\chi = \frac{1}{N}\frac{\partial\langle M\rangle}{\partial B} = \frac{\beta}{N}\langle(M-\langle M\rangle)^2\rangle\,, \quad (12.42)$$

where we see its relation to the fluctuations of the magnetization. This analysis can be repeated in a similar way for every pair of conjugate quantities.

## 12.4 Correlation Functions

The correlation functions can be obtained is a similar manner when we consider external fields which are space dependent. For simplicity, consider a system defined on a discrete lattice, whose sites are mapped to natural numbers $i = 1, \ldots, N$. Then the magnetic field $B_i$ is a function of the position $i$ and interacts with the spin $s_i$ so that

$$H = E - \sum_i B_i s_i\,. \quad (12.43)$$

Then the magnetization per site $m_i \equiv s_i$[14] at position $i$ is

$$\langle s_i\rangle = \frac{1}{\beta}\frac{\partial \ln Z}{\partial B_i}\,. \quad (12.44)$$

The connected two point correlation function is defined by

$$G_c^{(2)}(i,j) = \langle(s_i-\langle s_i\rangle)(s_j-\langle s_j\rangle)\rangle = \langle s_i s_j\rangle - \langle s_i\rangle\langle s_j\rangle = \frac{1}{\beta^2}\frac{\partial^2 \ln Z}{\partial B_i \partial B_j}\,. \quad (12.45)$$

When the values of $s_i$ and $s_j$ are strongly correlated, i.e. they "vary together" in the random samples that we take, the function (12.45) takes on large positive values. When the values of $s_i$ and $s_j$ are not at all correlated with each other, the terms $(s_i - \langle s_i\rangle)(s_j - \langle s_j\rangle)$ in the sum over $\mu$ in the expectation value $\langle(s_i - \langle s_i\rangle)(s_j - \langle s_j\rangle)\rangle$ cancel each other and $G_c^{(2)}(i,j)$ is zero[15].

[14] Actually the two quantities are proportional to each other, but for simplicity we set the proportionality constant equal to 1.

[15] There is also the possibility (not occurring in our discussion) that $s_i$ and $s_j$ are strongly anti-correlated in which case $G_c^{(2)}(i,j)$ is negative.

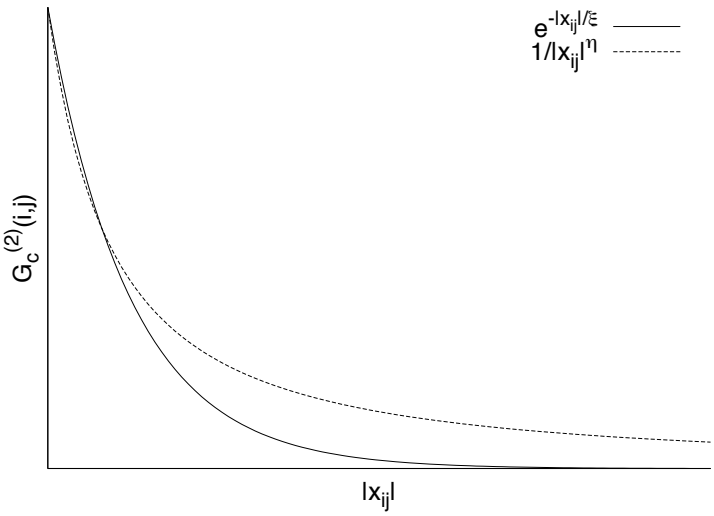



Figure 12.3: The connected two point correlation function $G_c^{(2)}(i,j)$ for $\xi < \infty$ and $\xi \to \infty$.

The function $G_c^{(2)}(i,j)$ takes its maximum value $\langle (s_i - \langle s_i \rangle)^2 \rangle$ for $i = j$. Then it falls off quite fast. For a generic system

$$G_c^{(2)}(i,j) \sim \mathrm{e}^{-|x_{ij}|/\xi}\,, \tag{12.46}$$

where $|x_{ij}|$ is the distance between the points $i$ and $j$. The *correlation length* $\xi$, is a characteristic length scale of the system which is a measure of the distance where there is a measurable correlation between the magnetic moments of two lattice sites. It depends on the parameters that define the system $\xi = \xi(\beta, B, N, \ldots)$. It is important to stress that it is a length scale that arises *dynamically*. In contrast, length scales like the size of the system $L$ or the lattice constant $a$ are parameters of the system which don't depend on the dynamics. In most of the cases, $\xi$ is of the order of a few lattice constants $a$ and such a system does not exhibit correlations at macroscopic scales (i.e. of the order of $L$).

Interesting physics arises when $\xi \to \infty$. This can happen by fine tuning the parameters on which $\xi$ depends on to their *critical values*. For example, in the neighborhood of a continuous[16] phase transition, the exponential falloff in (12.46) vanishes and $G_c^{(2)}(i,j)$ falls off like a power (see figure 12.3)

$$G_c^{(2)}(i,j) \sim \frac{1}{|x_{ij}|^{d-2+\eta}}\,, \tag{12.47}$$

[16] I.e. not of first order.

where $d$ is the number of dimensions of space and $\eta$ a *critical exponent*. As we approach the critical point[17], correlations extend to distances $|x_{ij}| \gg a$. Then the system is not sensitive to the short distance details of the lattice and its dynamics are very well approximated by continuum space dynamics. Then we say that we obtain the *continuum limit* of a theory which is microscopically defined on a lattice. Since the microscopic details become irrelevant, a whole class of theories with different microscopic definitions[18] have the same continuum limit. This phenomenon is called *universality* and plays a central role in statistical physics and quantum field theories.

## 12.5 Sampling

Our main goal is to calculate the expectation value $\langle \mathcal{O} \rangle$,

$$\langle \mathcal{O} \rangle = \sum_{\mu} p_\mu \, \mathcal{O}_\mu = \frac{\sum_\mu \mathcal{O}_\mu \mathrm{e}^{-\beta E_\mu}}{\sum_\mu \mathrm{e}^{-\beta E_\mu}} \,, \tag{12.48}$$

of a physical quantity, or *observable*, $\mathcal{O}$ of a statistical system in the canonical ensemble approximately. For this reason we construct a sample of $M$ states $\{\mu_1, \mu_2, \ldots, \mu_M\}$ which are distributed according to a chosen probability distribution $P_\mu$. We define the *estimator* $\mathcal{O}_M$ of $\langle \mathcal{O} \rangle$ to be

$$\mathcal{O}_M = \frac{\sum_{i=1}^M \mathcal{O}_{\mu_i} P_{\mu_i}^{-1} \mathrm{e}^{-\beta E_{\mu_i}}}{\sum_{i=1}^M P_{\mu_i}^{-1} \mathrm{e}^{-\beta E_{\mu_i}}} \,. \tag{12.49}$$

The above equation is easily understood since, for a large enough sample, $P_{\mu_i} \approx$ "*Frequency of finding* $\mu_i$ *in the sample*", and we expect that

$$\langle \mathcal{O} \rangle = \lim_{M \to \infty} \mathcal{O}_M \,. \tag{12.50}$$

Our goal is to find an appropriate $P_\mu$ so that the convergence of (12.50) is as fast as possible. Consider the following cases:

### 12.5.1 Simple Sampling

We choose $P_\mu =$ const., and equation (12.49) becomes

$$\mathcal{O}_M = \frac{\sum_{i=1}^M \mathcal{O}_{\mu_i} \mathrm{e}^{-\beta E_{\mu_i}}}{\sum_{i=1}^M \mathrm{e}^{-\beta E_{\mu_i}}} \,. \tag{12.51}$$

[17] If we tune many parameters, this is a *critical surface* in the parameter space.

[18] E.g. defined on square or triangular lattices, with nearest neighbor or next to nearest neighbor interactions.

The problem with this choice is the small *overlap* of the sample with the states that make the most important contributions to the sum in (12.48). As we have already mentioned in the introduction, the size of the sample in a Monte Carlo simulation is a minuscule fraction of the total number of states. Therefore, the probability of picking the ones that make important contributions to the sum in (12.48) is very small. Consider for example the case $\mathcal{O} = E$ in a generic model. According to equation (12.21) we have that

$$\langle E \rangle = \sum_E E\, p(E)\,, \tag{12.52}$$

where $p(E)$ is the probability of measuring energy $E$ in the system. A qualitative plot of $p(E)$ is shown in figure 12.1. From (12.25) and (12.28) we have that $E^* \sim 1/\beta$ and $\Delta E \sim 1/\beta$, therefore for $\beta = 0$ and $\beta > 0$ the qualitative behavior of the respective $p(E)$ distributions is shown in figure 12.4. The distribution of the simple sampling corresponds to the case $\beta =$

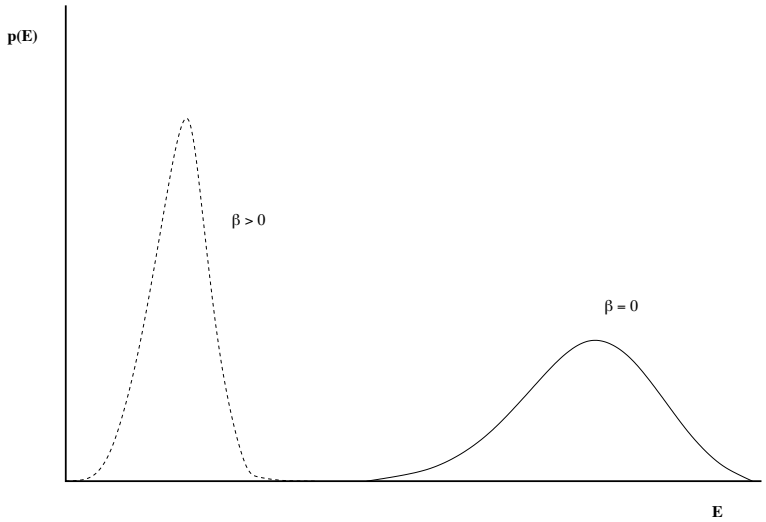



Figure 12.4: The distributions $p(E)$ for a generic model for temperatures $\beta = 0$ and $\beta > 0$. The two distributions have negligible overlap. In order that the $\beta = 0$ distribution is as shown, we assume that the energy of all states is bounded and that the system has a finite number of degrees of freedom.

0 in equation (12.4), since $p_\mu$ =const. in this case[19]. In order to calculate the sum (12.52) with acceptable accuracy for $\beta > 0$ we have to obtain

[19]For these statements to be well defined, we assume that the energy of all states is bounded and that the system has a finite number of degrees of freedom. Otherwise consider the overlap for two temperatures $\beta_1 \gg \beta_2$.

a good sample in the region where the product $E\,p_{\beta>0}(E)$ is relatively important. The probability of obtaining a state such that $E\,p_{\beta>0}(E)$ is non negligible is very small when we use the $p_{\beta=0}(E)$ distribution. This can be seen pictorially in figure 12.4.

Even though this method has this serious shortcoming, it could still be useful in some cases. We have already applied it in the study of random walks. Note that, by applying equation (12.51), we can use the same sample for calculating expectation values for all values of $\beta$.

### 12.5.2 Importance Sampling

From the previous discussion it has become clear that, for a large system, a very small fraction of the space of states makes a significant contribution to the calculation of $\langle\mathcal{O}\rangle$. If we choose a sample with probability

$$P_\mu = p_\mu = \frac{\mathrm{e}^{-\beta E_\mu}}{Z}\,, \tag{12.53}$$

then we expect to sample exactly within this region. Indeed, the estimator, given by equation (12.49), is calculated from

$$\mathcal{O}_M = \frac{\sum_{i=1}^{M} \mathcal{O}_{\mu_i} \left(\mathrm{e}^{-\beta E_{\mu_i}}\right)^{-1} \mathrm{e}^{-\beta E_{\mu_i}}}{\sum_{i=1}^{M} \left(\mathrm{e}^{-\beta E_{\mu_i}}\right)^{-1} \mathrm{e}^{-\beta E_{\mu_i}}} = \frac{1}{M}\sum_{i=1}^{M} \mathcal{O}_{\mu_i}\,. \tag{12.54}$$

Sampling this way is called *importance sampling*, and it is the method of choice in most Monte Carlo simulations. The sample depends on the temperature $\beta$ and the calculation of the expectation values (12.54) requires a new sample for each[20] $\beta$. This extra effort, however, is much smaller than the one required in order to overcome the overlap problem discussed in the previous subsection.

## 12.6 Markov Processes

Sampling according to a desired probability distribution $P_\mu$ is not possible in a direct way. For example, if we attempt to construct a sample according to $P_\mu = \frac{\mathrm{e}^{-\beta E_\mu}}{Z}$ by picking a state $\mu$ by chance and add it to the sample with probability $P_\mu$, then we have a very small probability to accept that state in the sample. Therefore, the difficulty of constructing the sample

[20] We can use the same sample for a range of temperatures by using the *histogram method*, see [4].

runs into the same overlap problem as in the case of simple sampling. For this reason we construct a *Markov chain* instead. The members of the sequence of the chain will be our sample. A Markov process, or a Markov chain, is a stochastic process which, given the system in a state $\mu$, puts the system in a new state $\nu$ in such a way that it has the *Markov property*, i.e. that it is memoryless. This means that a chain of states

$$\mu_1 \to \mu_2 \to \ldots \to \mu_M \,, \tag{12.55}$$

is constructed in such a way that the *transition probabilities* $P(\mu \to \nu)$ from the state $\mu$ to a new state $\nu$ satisfy the following requirements:

1. They are independent of "time"
2. They depend only on the states $\mu$ and $\nu$ and not on the path that the system has followed on order to get to the sate $\mu$ (memorylessness)
3. The relation
$$\sum_\nu P(\mu \to \nu) = 1 \tag{12.56}$$
holds. Beware, in most of the cases $P(\mu \to \mu) > 0$, i.e. the system has a nonzero probability to remain in the same state
4. For $M \to \infty$ the sample $\{\mu_i\}$ follows the $P_\mu$ distribution.

Then our sample will be $\{\mu_i\} \equiv \{\mu_1, \mu_2, \ldots, \mu_M\}$. We may imagine that this construction happens in "time" $i = 1, 2, \ldots, M$. In a Monte Carlo simulation we construct a sample from a Markov chain by appropriately choosing the transition probabilities $P(\mu \to \nu)$ so that the convergence 4. is fast.

Choosing the initial state $\mu_1$ can become a non trivial task. If it turns out not to be a typical state of the sample, then it could take a long "time" for the system to "equilibrate", i.e. for the Markov process to start sampling states typical of the simulated temperature. The required time for this to happen is called the *thermalization time* which can become a serious part of our computational effort if we make a wrong choice of $\mu_1$ and/or $P(\mu \to \nu)$.

A necessary condition for the sample to converge to the desired distribution is for the process to be *ergodic*. This means that for every state $\mu$ it is possible to reach any other state $\nu$ in a finite number of steps. If this criterion is not satisfied and a significant part of phase space is not sampled, then sampling will fail. Usually, given a state $\mu$, the reachable states $\nu$ at the next step (i.e. the states for which $P(\mu \to \nu) > 0$) are

very few. Therefore the ergodicity of the algorithm considered must be checked carefully[21].

## 12.7 Detailed Balance Condition

Equation (12.2) tells us that, in order to find the system in equilibrium in the $p_\mu$ distribution, the transition probabilities should be such that

$$\sum_\nu p_\mu P(\mu \to \nu) = \sum_\mu p_\nu P(\nu \to \mu)\,. \tag{12.57}$$

This means that the rate that the system comes into the state $\mu$ is equal to the rate in which it leaves $\mu$. From equation (12.56) we obtain

$$p_\mu = \sum_\mu p_\nu P(\nu \to \mu)\,. \tag{12.58}$$

This condition is necessary but it is not sufficient (see section 2.2.3 in [4]). A sufficient, but not necessary, condition is the *detailed balance condition*. When the transition probabilities satisfy

$$p_\mu P(\mu \to \nu) = p_\nu P(\nu \to \mu)\,, \tag{12.59}$$

then the system will equilibrate to $p_\mu$ after sufficiently long thermalization time. By summing both sides of (12.59), we obtain the equilibrium condition (12.57). For the canonical ensemble (12.4) the condition becomes

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \frac{p_\nu}{p_\mu} = \mathrm{e}^{-\beta(E_\nu - E_\mu)}\,. \tag{12.60}$$

One can show that if the transition probabilities satisfy the above conditions then the equilibrium distribution of the system will be the Boltzmann distribution (12.4). A program implementing a Monte Carlo simulation of a statistical system in the canonical ensemble consists of the following main steps:

[21]There exist algorithms which are non-ergodic but the non reachable states are of "measure zero" in the space of states. These algorithms are formally non ergodic, but they are ergodic from a practical point of view. On the contrary, there exist algorithms that are formally ergodic but there are large regions of phase space where the probability of getting there is very small. This puts "ergodic barriers" in the sampling which will lead to wrong results. A common example is sampling a system in the neighborhood of a first order phase transition where, for large systems, it is very hard to sample states in both phases.

1. Write a program that codes appropriately chosen transition probabilities $P(\mu \to \nu)$ that satisfy condition (12.60)
2. Choose an initial state $\mu_1$
3. Let the system evolve until it thermalizes to the Boltzmann distribution (12.4) (thermalization)
4. Collect data for the observables $\mathcal{O}$ and calculate the estimators $\mathcal{O}_M$ from equation (12.54)
5. Stop when the desired accuracy in the calculation of $\langle \mathcal{O} \rangle$ has been achieved.

Equation (12.60) has many solutions. For a given problem, we are looking for the most efficient one. Below we list some possible choices:

$$P(\mu \to \nu) = A \cdot \mathrm{e}^{-\frac{1}{2}\beta(E_\nu - E_\mu)}\,, \tag{12.61}$$

$$P(\mu \to \nu) = A \cdot \frac{\mathrm{e}^{-\beta(E_\nu - E_\mu)}}{1 + \mathrm{e}^{-\beta(E_\nu - E_\mu)}}\,, \tag{12.62}$$

$$P(\mu \to \nu) = A \cdot \begin{cases} \mathrm{e}^{-\beta(E_\nu - E_\mu)} & E_\nu - E_\mu > 0 \\ 1 & E_\nu - E_\mu \leq 0 \end{cases}\,, \tag{12.63}$$

for appropriately chosen states $\nu \neq \mu$ and

$$P(\mu \to \mu) = 1 - \sum_\nu P(\mu \to \nu)\,. \tag{12.64}$$

$P(\mu \to \nu') = 0$ for any other state $\nu'$. In order for (12.64) to be meaningful, the constant $A$ has to be chosen so that

$$\sum_{\nu \neq \mu} P(\mu \to \nu) < 1\,. \tag{12.65}$$

Equation (12.65) gives much freedom in the choice of transition probabilities. In most cases, we split $P(\mu \to \nu)$ in two independent parts

$$P(\mu \to \nu) = g(\mu \to \nu)\, A(\mu \to \nu)\,. \tag{12.66}$$

The probability $g(\mu \to \nu)$ is the *selection probability* of the state $\nu$ when the system is in the state $\mu$. Therefore the first step in the algorithm is to select a state $\nu \neq \mu$ with probability $g(\mu \to \nu)$.

The second step is to accept the change with probability $A(\mu \to \nu)$. If the answer is no, then the system remains in the state $\mu$. This way equation (12.64) is satisfied. The probabilities $A(\mu \to \nu)$ are called the *acceptance ratios*.

The art in the field is to device algorithms that give the maximum possible acceptance ratios for the new states $\nu$ and that the states $\nu$ are as much as possible statistically independent from the original state $\mu$. An ideal situation is to have $A(\mu \to \nu) = 1$ for all $\nu$ for which $g(\mu \to \nu) > 0$. As we will see in a following chapter, this is what happens in the case of the Wolff cluster algorithm.

## 12.8 Problems

1. Prove equation (12.18).
2. Prove equation (12.19).
3. Prove equation (12.45).
4. Show that equations (12.61)–(12.63) satisfy (12.60).

# Chapter 13

# Simulation of the $d = 2$ Ising Model

This chapter is an introduction to the basic Monte Carlo methods used in the simulations of the Ising model on a two dimensional rectangular lattice, but also in a wide spectrum of scientific applications. We will introduce the Metropolis algorithm, which is the most common algorithm used in Monte Carlo simulations. We will discuss the thermalization of the system and the effect of correlations between successive spin configurations generated during the simulation. The *autocorrelation function* and the time scale defined by it, the *autocorrelation time*, are measures of these autocorrelations and play a central role in the study of the statistical independence of our measurements. Beating autocorrelations is crucial in Monte Carlo simulations since they are the main obstacle for studying large systems, which in turn is essential for taking the thermodynamic limit without the systematic errors introduced by finite size effects. We will also introduce methods for the computation of statistical errors that take into account autocorrelations. The determination of statistical errors is of central importance in order to assess the quality of a measurement and predict the amount of resources needed for reaching a specific accuracy goal.

## 13.1 The Ising Model

The Ising model (1925) [52] has played an important role in the evolution of ideas in statistical physics and quantum field theory. In particular, the two dimensional model is complicated enough in order to possess nontrivial properties but simple enough in order to be able to

obtain an exact analytic solution. The zero magnetic field model has a 2nd order phase transition for a finite value of the temperature and we are able to compute critical exponents and study its continuum limit in detail. This gives us valuable information on the non analytic properties of a system undergoing a second order phase transition, the appearance of scaling, the renormalization group and universality. Using the exact solution[1] of Onsager (1948) [53] and others, we obtain exact results and compare them with those obtained via approximate methods, like Monte Carlo simulations, high and low temperature expansions, mean field theory etc. The result is also interesting from a physics point of view, since it is the simplest, phenomenologically interesting, model of a ferromagnetic material. Due to universality, the model describes also the liquid/vapor phase transition at the triple point. A well known textbook for a discussion of statistical mechanical models that can be solved exactly is the book by Baxter [51].

In order to define the model, consider a two dimensional square lattice like the one shown in figure 13.1. On each *site* or *node* of the lattice we have an "atom" or a "magnet" of spin $s_i$. The geometry is determined by the distance of the nearest neighbors, the *lattice constant* $a$, and the number of sites $N$. Each side consists of $L$ sites so that $N = L \times L = L^d$, where $d = 2$ is the dimension of space. The topology is determined by the way sites are connected with each other via links. Special care is given to the sites located on the sides of the lattice. We usually take *periodic boundary conditions* which is equivalent to identifying the opposite sides of the square by connecting their sites with a link. This is depicted in figure (13.1). Periodic boundary conditions endow the plane on which the lattice is defined with a toroidal topology. The system's dynamics are determined by the spin–spin interaction. We take it to be *short range* and the simplest case considered here takes into account only nearest neighbor interactions. In the Ising model, spins have two possible values, "up" or "down" which we map[2] to the numerical values $+1$ or $-1$. For the ferromagnetic model, each link is a "bond" whose energy is higher when the spins on each side of the link are pointing in the same direction and lower when they point in the opposite[3] direction. This is depicted in figure 13.1. The system could also be immersed in a constant magnetic

[1]For a very nice proof of Onsager's solution look at the book by T. Huang [54] and the paper by C.N. Yang [55].

[2]This is only a convention. We could have picked 0 and 1 or any other pair of labels. The choice of labels affects only the expression of the Hamiltonian and related observables.

[3]The opposite is true for the antiferromagnetic model.

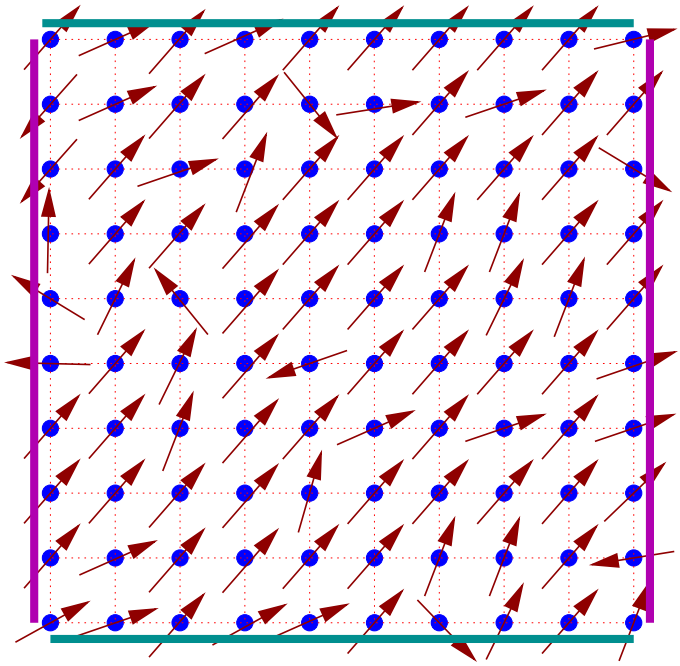


Figure 13.1: The two dimensional square lattice whose sites $i = 1, \ldots, N$ are occupied by "atoms" or "magnets" with spin $s_i$. In this figure spins may have any orientation on the plane (XY model). The simplest models take into account only the nearest neighbor interactions $-J\vec{s}_i \cdot \vec{s}_j$, where $\langle ij \rangle$ is a *link* of the lattice. We take periodic boundary conditions which result in a toroidal topology on the lattice where the horizontal and vertical sides of the lattice are identified. In the figure, identified sides have the same color and their respective sites are connected by a link..

field $B$ whose direction is parallel to the direction of the spins.

We are now ready to write the Hamiltonian and the partition function of the system. Consider a square lattice of $N$ lattice sites (or *vertices*) labeled by a number $i = 1, 2, \ldots, N$. The lattice has $N_l$ links (or *bonds*) among nearest neighbors. These are labeled by $\langle ij \rangle$, where $(i, j)$ is the pair of vertices on each side of the link. We identify the sides of the square like in figure 13.1. Then, since two vertices are connected by one link and four links intersect at one vertex, we have that[4]

$$2N_l = 4N \Rightarrow N_l = 2N\,. \tag{13.1}$$

At each vertex we place a spin $s_i = \pm 1$. The Hamiltonian of the system

[4]It is easy to see that each vertex is in a one to one correspondence with a pair of links, say the east and north bound ones.

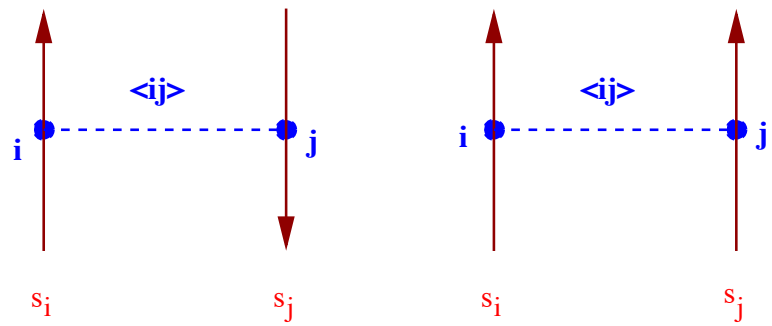



Figure 13.2: The Ising model spins take two possible values: "up" or "down" and the Hamiltonian of the system is the sum of contributions of the energy of all links ("bonds") $\langle ij \rangle$. The energy of each bond takes two values, $+J$ for opposite or $-J$ for same spins, where $J > 0$ for a ferromagnetic system. The system possesses a discrete $Z_2$ symmetry: The Hamiltonian is invariant when all $s_i \to -s_i$.

is given by

$$H = -J \sum_{\langle ij \rangle} s_i s_j - B \sum_i s_i \,. \tag{13.2}$$

The first term is the spin–spin interaction and for $J > 0$ the system is ferromagnetic. In this book, we consider only the $J > 0$ case. A link connecting same spins has energy $-J$, whereas a link connecting opposite spins has energy $+J$. The difference of the energy between the two states is $2J$ and the spin-spin dynamics favor links connecting same spins. The minimum energy $E_0$ is obtained for the *ground state*, which is the *unique*[5] state in which all spins point in the direction[6] of $B$. This is equal to

$$E_0 = -JN_l - BN = -(2J + B)N \,. \tag{13.3}$$

The partition function is

$$Z = \sum_{s_1=\pm 1} \sum_{s_2=\pm 1} \cdots \sum_{s_N=\pm 1} \mathrm{e}^{-\beta H[\{s_i\}]} \equiv \sum_{\{s_i\}} \mathrm{e}^{\beta J \sum_{\langle ij \rangle} s_i s_j + \beta B \sum_i s_i} \,, \tag{13.4}$$

where $\{s_i\} \equiv \{s_1, s_2, \ldots, s_N\}$ is a spin configuration of the system. The number of terms is equal to the number of configurations $\{s_i\}$, which is

[5]When $B = 0$ the system has an "up–down" $Z_2$ symmetry. This means that states connected by the transformation $s_i \to -s_i$ for all $i$ result in the same Hamiltonian. In this case we have two ground states and the system chooses one of them by *spontaneously breaking* the $Z_2$ symmetry.

[6]The vacuum structure of the antiferromagnetic system $J < 0$ for $B = 0$ is much richer.

equal to $2^N$, i.e. it increases exponentially with $N$. For a humble $5 \times 5$ lattice we have $2^{25} \approx 3.4 \times 10^6$ terms.

The two dimensional Ising model for $B = 0$ has the interesting property that, for $\beta = \beta_c$, where

$$\beta_c = \frac{1}{2} \ln (1 + \sqrt{2}) \approx 0.4406867935 \ldots , \tag{13.5}$$

it undergoes a phase transition between an *ordered* or *low temperature* phase where the system is magnetized ($\langle |M| \rangle > 0$) and a *disordered* or *high temperature* phase where the magnetization vanishes ($\langle |M| \rangle = 0$). The magnetization $\langle |M| \rangle$ distinguishes between the two phases and it is called the *order parameter*. The critical temperature $\beta_c$ is the Curie temperature. The phase transition is of second order, which is a special case of a *continuous phase transition*. For a continuous phase transition the order parameter is continuous at $\beta = \beta_c$, but it is non analytic[7]. For a second order phase transition, its derivative is not continuous. This is qualitatively depicted in figure 13.1.

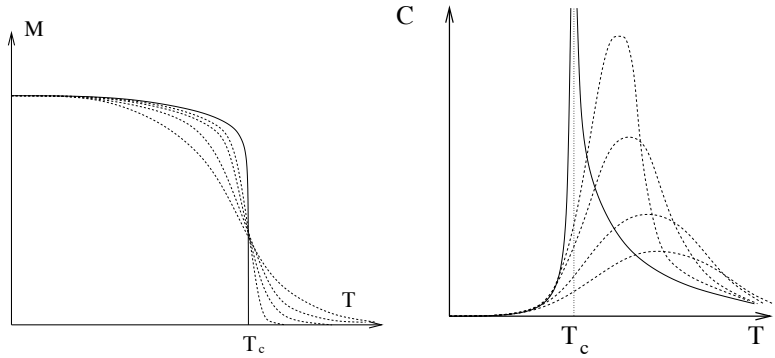



Figure 13.3: The qualitative behavior of the magnetization (left) and the specific heat (right) near the Ising model phase transition. The continuous line is the non analytic behavior in the thermodynamic limit, whereas the dashed lines show the behavior of the analytic, finite $N$ behavior. The latter converge to the former in the large $N$ limit (thermodynamic limit).

For $\beta \neq \beta_c$ the correlation function (12.45) behaves like in equation (12.46) resulting in a finite correlation length $\xi(\beta)$. The correlation length[8] diverges as we approach the critical temperature, and its asymp-

[7]In contrast, a first order phase transition is a transition where the order parameter itself is discontinuous.

[8]We mean the correlation length in the thermodynamic limit, i.e. we take the large $N$ limit *first*.

totic behavior in this limit is given by the scaling relation

$$\xi(\beta) \equiv \xi(t) \sim |t|^{-\nu}\,, \qquad t = \frac{\beta_c - \beta}{\beta_c}\,. \tag{13.6}$$

Then the correlation function behaves according to (12.47)

$$G_c^{(2)}(i,j) \sim \frac{1}{|x_{ij}|^{\eta}}\,. \tag{13.7}$$

Scaling behavior is also found for the specific heat $C$, the magnetization $M \equiv \langle |M| \rangle$ and the magnetic susceptibility $\chi$ according to the relations

$$C \sim |t|^{-\alpha} \tag{13.8}$$
$$M \sim |t|^{\beta} \tag{13.9}$$
$$\chi \sim |t|^{-\gamma}\,, \tag{13.10}$$

whereas the magnetization for $t = 0$ and nonzero magnetic field $B \neq 0$ behaves like

$$M \sim B^{-1/\delta}\,. \tag{13.11}$$

The exponents in the above scaling relations are called *critical exponents* or *scaling exponents*. They take universal values, i.e. they don't depend on the details of the lattice construction or of the interaction. A whole class of such models with different microscopic definitions have the exact same long distance behavior[9]! The systems in the same *universality class* need to share the same symmetries and dimensionality of space and the fact that the interaction is of short range. In the particular model that we study, these exponents take the so called *Onsager exponent* values

$$\begin{aligned} \alpha = 0\,, \quad \beta = \tfrac{1}{8}\,, \quad \gamma = \tfrac{7}{4} \\ \delta = 15\,, \quad \nu = 1\,, \quad \eta = \tfrac{1}{4}\,. \end{aligned} \tag{13.12}$$

Theses exponents determine the non analytic behavior of the corresponding functions in the thermodynamic limit. Non analyticity cannot arise in the finite $N$ model. The partition function (13.4) is a sum of a finite number of analytic terms, which of course result in an analytic function. The non analytic behavior manifests in the $N \to \infty$ limit, where the finite $N$ analytic functions converge to a non analytic one. The loss of analyticity is related to the appearance of long distance correlations

[9] i.e. at distances larger than the (diverging) correlation length.

between the spins and the scaling of the correlation length according to equation (13.6).

The two phases, separated by the phase transition, are identified by the different values of an order parameter. Each phase is characterized by the appearance or the breaking of a symmetry. In the Ising model, the order parameter is the magnetization and the symmetry is the $Z_2$ symmetry represented by the transformation $s_i \to -s_i$. The magnetization is zero in the disordered, high temperature phase and non zero in the ordered, low temperature phase. This implies that the magnetization is a non analytic function of the temperature[10].

*Universality* and *scale invariance* appear in the $\xi \to \infty$ limit. In our case, this occurs by tuning only one parameter, the temperature, to its critical value. A unique, dynamical, length scale emerges from the correlation function, the correlation length $\xi$. Scale invariance manifests when the correlation length becomes much larger than the microscopic length scale $a$ when $\beta \to \beta_c$. In the critical region, all quantities which are functions of the distance become functions only of the ratio $r/\xi$. Everything depends on the long wavelength fluctuations required by the symmetry of the order parameter and all models in the same universality class have the same long distance behavior. This way one can study only the simplest model within a universality class in order to deduce the large distance/long wavelength properties of all systems in the class.

## 13.2 Metropolis

Consider a square lattice with $L$ sites on each side so that $N = L \times L = L^2$ is the number of lattice sites (vertices) and $N_l = 2N$ is the number of links (bonds) between the sites. The relation $N_l = 2N$ holds because we choose helical boundary conditions as shown in figure 13.6. The choice of boundary conditions will be discussed later. On each site $i$ we have one degree of freedom, the "spin" $s_i$ which takes on two values $\pm 1$. We consider the case of zero magnetic field $B = 0$, therefore the Hamiltonian is given by[11]

$$H = -\sum_{\langle ij \rangle} s_i s_j \,. \tag{13.13}$$

[10] An analytic function which is zero in an arbitrarily small interval, it is - by Taylor expanding around a point in this interval - everywhere zero.

[11] The constant $J = 1$ by choosing appropriate units for the $s_i$.

The sum $\sum_{\langle ij\rangle}$ is a sum over the *links* $\langle ij\rangle$, corresponding to the pairs of sites $i, j$. Then $\sum_{\langle i,j\rangle} = (1/2)\sum_{i=1}^{N}\sum_{j=1}^{N}$ since each bond is counted twice in the second sum. The partition function is

$$Z = \sum_{s_1=\pm 1}\sum_{s_2=\pm 1}\cdots\sum_{s_N=\pm 1} \mathrm{e}^{-\beta H[\{s_i\}]} \equiv \sum_{\{s_i\}} \mathrm{e}^{\beta\sum_{\langle ij\rangle} s_i s_j}\,. \tag{13.14}$$

Our goal is to collect a sample of states that is distributed according to the Boltzmann distribution (12.4). This will be constructed via a Markov process according to the discussion in section 12.6. Sampling is made according to (12.53) and the expectation values are estimated from the sample using (12.54). At each step the next state is chosen according to (12.60), and for large enough sample, or "time steps", the sample is approximately in the desired distribution.

Suppose that the system is in a state[12] $\mu$. According to (12.66), the probability that in the next step the system goes into the state $\nu$ is

$$P(\mu\to\nu) = g(\mu\to\nu)\,A(\mu\to\nu)\,, \tag{13.15}$$

where $g(\mu\to\nu)$ is the *selection probability* of the state $\nu$ when the system is in the state $\mu$ and $A(\mu\to\nu)$ is the *acceptance ratio*, i.e. the probability that the system jumps into the new state. If the detailed balance condition (12.60)

$$\frac{P(\mu\to\nu)}{P(\nu\to\mu)} = \frac{g(\mu\to\nu)\,A(\mu\to\nu)}{g(\nu\to\mu)\,A(\nu\to\mu)} = \mathrm{e}^{-\beta(E_\nu - E_\mu)} \tag{13.16}$$

is satisfied, then the distribution of the sample will converge to (12.4) $p_\mu = \mathrm{e}^{-\beta E_\mu}/Z$. In order that the system changes states often enough, the probabilities $P(\mu\to\nu)$ should be of order one and the differences in the energy $E_\nu - E_\mu$ should not be too large. This means that the product of the temperature with the energy difference should be a number of order one or less. One way to accomplish this is to consider states that differ by the value of the spin on only one site $s_i = \pm 1 \to s_i' \mp 1$. Since the energy (13.13) is a local quantity, the change in energy will be small. More specifically, if each site has $z = 4$ nearest neighbors, the change of the spin on site $i$ results in a change of sign for $z$ terms $s_i s_j$ in (13.13). The change in the energy for each bond is $\pm 2$. If the state $\mu$ is given by $\{s_1, \ldots, s_i, \ldots, s_N\}$ and the state $\nu$ by $\{s_1, \ldots, s_i', \ldots, s_N\}$ (i.e. all the spins are the same except the spin $s_i$ which changes sign), the energy difference will be

$$|\Delta E| \le 2z \Leftrightarrow E_\mu - 2z \le E_\nu \le E_\mu + 2z\,. \tag{13.17}$$

[12]The state $\mu$ is determined by a spin configuration $\{s_i\}_{i=1\ldots N}$.

If the site $i$ is randomly chosen then

$$g(\mu \to \nu) = g(\nu \to \mu) = \begin{cases} \frac{1}{N} & (\mu, \nu) \text{ differ by one spin} \\ 0 & \text{otherwise} \end{cases} , \tag{13.18}$$

and the algorithm is ergodic. Then we have that

$$\frac{A(\mu \to \nu)}{A(\nu \to \mu)} = \mathrm{e}^{-\beta(E_\nu - E_\mu)} . \tag{13.19}$$

A simple choice for satisfying this condition is (12.61)

$$A(\mu \to \nu) = A_0 \cdot \mathrm{e}^{-\frac{1}{2}\beta(E_\nu - E_\mu)} . \tag{13.20}$$

In order to maximize the acceptance ratios we have to take $A_0 = \mathrm{e}^{-\beta z}$. Remember that we should have $A(\mu \to \nu) \leq 1$ and $|\Delta E| \leq 2z$. Therefore

$$A(\mu \to \nu) = \mathrm{e}^{-\frac{1}{2}\beta(E_\nu - E_\mu + 2z)} . \tag{13.21}$$

Figure 13.4 depicts the dependence of $A(\mu \to \nu)$ on the change in energy

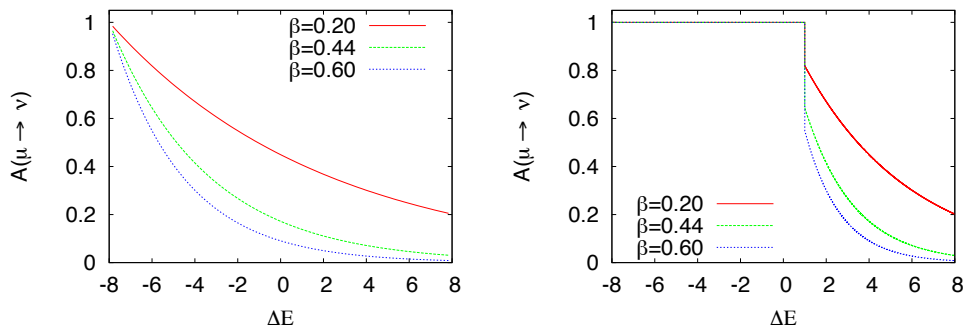



Figure 13.4: The acceptance ratio $A(\mu \to \nu)$ for the two dimensional Ising model on a square lattice given by equation (13.21) (left) and the Metropolis algorithm (right) as a function of the change in energy $\Delta E = E_\nu - E_\mu$. For the Metropolis algorithm the acceptance ratios are larger and the algorithm is expected to perform better.

for different values of $\beta$. We observe that this probability is small even for zero energy change and we expect this method not to perform very well.

It is much more efficient to use the algorithm proposed by Nicolas Metropolis *et. al.* 1953 [56] which is given by (12.63)

$$A(\mu \to \nu) = \begin{cases} \mathrm{e}^{-\beta(E_\nu - E_\mu)} & E_\nu - E_\mu > 0 \\ 1 & E_\nu - E_\mu \leq 0 \end{cases} . \tag{13.22}$$

According to this relation, when a change in the states lowers the energy, the change is always accepted. When it increases the energy, the change is accepted with a probability less than one. As we can see in figure 13.4, this process accepts new states much more frequently than the previous algorithm.

The Metropolis algorithm is very widely used. It is applicable to any system, it is simple and efficient. We note that the choice to change the spin only locally is not a restriction put by the metropolis algorithm. There exist efficient algorithms that make non local changes to the system's configuration that (almost) conserve the Hamiltonian[13] and, consequently, the acceptance ratios are satisfactorily large.

## 13.3 Implementation

The first step in designing a code is to define the data structure. The degrees of freedom are the spins $s_i = \pm 1$ which are defined on $N$ lattice sites. The most important part in designing the data structure in a lattice simulation is to define the neighboring relations among the lattice sites in the computer memory and this includes the implementation of the boundary conditions. A bad choice of boundary conditions will make the effect of the boundary on the results to be large and increase the *finite size effects*. This will affect the speed of convergence of the results to the thermodynamic limit, which is our final goal. The most popular choice is the toroidal or periodic boundary conditions. A small variation of these lead to the so called *helical* boundary conditions, which will be our choice because of their simplicity. Both choices share the fact that each site has the same number of nearest neighbors, which give the same local geometry everywhere on the lattice and minimize finite size effects due to the boundary. In contrast, if we choose fixed or free boundary conditions on the sides of the square lattice, the boundary sites have a smaller number of nearest neighbors than the ones inside the lattice.

One choice for mapping the lattice sites into the computer memory is to use their coordinates $(i, j)$, $i, j = 1, \ldots, L$. Each spin is stored in an array `s(L,L)`. For a site `s(i,j)` the four nearest neighbors are `s(i±1,j)`, `s(i,j±1)`. The periodic boundary conditions are easily implemented by adding $\pm L$ to `i,j` each time they become less than one or greater than $L$. This is shown in figures 13.5 and 13.34.

The elements of the array `s(L,L)` are stored linearly into the com-

[13] An example is the Hybrid Monte Carlo used in lattice QCD simulations.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 | 6 | 7 | 8 | 9 | 10 | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | 11 | 12 | 13 | 14 | 15 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 16 | 17 | 18 | 19 | 20 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 21 | 22 | 23 | 24 | 25 | 21 | 22 | 23 | 24 | 25 |
| 1 | 2 | 3 | 4 | 5 | **1** | **2** | **3** | **4** | **5** | 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 | **6** | **7** | **8** | **9** | **10** | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | **11** | **12** | **13** | **14** | **15** | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | **16** | **17** | **18** | **19** | **20** | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | **21** | **22** | **23** | **24** | **25** | 21 | 22 | 23 | 24 | 25 |
| 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 | 6 | 7 | 8 | 9 | 10 | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | 11 | 12 | 13 | 14 | 15 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 16 | 17 | 18 | 19 | 20 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 21 | 22 | 23 | 24 | 25 | 21 | 22 | 23 | 24 | 25 |

Figure 13.5: An $L = 5$ square lattice with periodic boundary conditions. The topology is toroidal.

puter memory. The element `s(i,j)` is at a "distance" `(j-1)*L+i` array positions from `s(1,1)` and accessing its value involves an, invisible to the programmer, multiplication. Using helical boundary conditions this multiplication can be avoided. The positions of the lattice sites are now given by one number $i = 1, \ldots, L^2 = N$, as shown in figures 13.6 and 13.35. The spins are stored in memory in a one dimensional array `s(N)` and the calculation of the nearest neighbors of a site `s(i)` is easily done by taking the spins `s(i±1)` and `s(i±L)`. The simplicity of the helical boundary conditions is based on the fact that, for the nearest neighbors of sites on the sides of the square, all we have to do is to make sure that the index `i` stays within the accepted range `1≤ i ≤ N`. This is easily done by adding or subtracting $N$ when necessary. Therefore in a program that we want to calculate the four nearest neighbors `nn` of a site `i`, all we have to do is:

```
nn=i+1;if(nn.gt.N)nn=nn-N
nn=i-1;if(nn.lt.1)nn=nn+N
nn=i+L;if(nn.gt.N)nn=nn-N
nn=i-L;if(nn.lt.1)nn=nn+N
```

We will choose helical boundary conditions for their simplicity and effi-

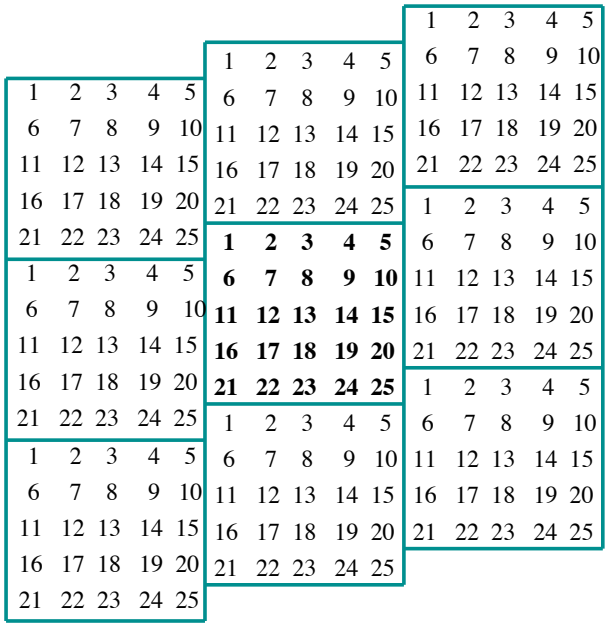



Figure 13.6: An $L = 5$ square lattice with helical boundary conditions. The topology is toroidal.

ciency in calculating nearest neighbors[14].

The dynamics of the Monte Carlo evolution is determined by the initial state and the Metropolis algorithm. A good choice of initial configuration can be important in some cases. It could lead to fast or slow thermalization, or even to no thermalization at all. In the model that we study it will not play an important role, but we will discuss it because of its importance in the study of other systems. We may choose a "cold" ($\beta = +\infty$ - all spins aligned) or a "hot" ($\beta = 0$ - all spins are equal to $\pm 1$ with equal probability $1/2$) initial configuration. For large lattices, it is desirable to start in one of these states and then lower/increase the temperature in small steps. Each time that the temperature is changed, the spin configuration is saved and used in the next simulation.

Ergodicity and thermalization must be checked by performing independent simulations[15] and verify that we obtain the same results. Simi-

[14]On the bad side, helical boundary conditions introduce a small finite size effect due to the shift of lattice positions in neighboring copies of the lattice. If one has to study small lattices, especially in higher dimensions, the best choice is to use periodic boundary conditions. We are going to study large enough lattices that this finite size effect is negligible.

[15]Different sequence of random numbers.

larly, independent simulations starting from different initial states must also be checked that yield the same results.

Consider each step in the Markov process defined by the Metropolis algorithm. Assume that the system is in the state $\mu = \{s_1^\mu, \ldots, s_k^\mu, \ldots, s_N^\mu\}$ and consider the transition to a new state $\nu = \{s_1^\nu, \ldots, s_k^\nu, \ldots, s_N^\nu\}$ which differs only by the value of the spin $s_k^\nu = -s_k^\mu$ (spin flip), whereas all the other spins are the same: $s_j^\nu = s_j^\mu \ \forall j \neq k$. The energy difference between the two states is

$$\begin{aligned} E_\nu - E_\mu &= (-\sum_{\langle ij \rangle} s_i^\nu s_j^\nu) - (-\sum_{\langle ij \rangle} s_i^\mu s_j^\mu) \\ &= -\sum_{\langle ik \rangle} s_i^\mu (s_k^\nu - s_k^\mu) \\ &= 2\sum_{\langle ik \rangle} s_i^\mu s_k^\mu \\ &= 2 s_k^\mu \left( \sum_{\langle ik \rangle} s_i^\mu \right) , \end{aligned} \tag{13.23}$$

where the second line is obtained after the cancellation of the common terms in the sums. In the third line we used the relation $s_k^\nu - s_k^\mu = -2s_k^\mu$, which you can prove easily by examining the cases $s_k^\mu = \pm 1$ separately. The important property of this relation is that it is local since it depends only on the nearest neighbors. The calculation of the energy difference $E_\nu - E_\mu$ is fast and is always a number of order one[16].

The Metropolis condition is easily implemented. We calculate the sum in the parenthesis of the last line of equation (13.23) and obtain the energy difference $E_\nu - E_\mu$. If the energy decreases, i.e. $E_\nu - E_\mu \leq 0$, the new state $\nu$ is accepted and "we flip the spin". If the energy increases, i.e. $E_\nu - E_\mu > 0$, then the acceptance ratio is $A(\mu \to \nu) = \mathrm{e}^{-\beta(E_\nu - E_\mu)} < 1$. In order to accept the new state with this probability we pick a random number uniformly distributed in $0 \leq x < 1$. The probability that this number is $x < A(\mu \to \nu)$ is equal to[17] $A(\mu \to \nu)$. Therefore if $x \leq A(\mu \to \nu)$ the change is accepted. If $x > A(\mu \to \nu)$ the change is rejected and the system remains in the same state $\mu$.

A small technical remark is in order: The possible values of the sum $\left(\sum_{\langle ik \rangle} s_i^\mu\right) = -4, -2, 0, 2, 4$ and these are the only values that enter in the calculation of $A(\mu \to \nu)$. Moreover, only the values that increase

[16] An important fact is that it does not increase with the system size.

[17] For the uniform distribution $P(x < a) = a$.

the energy, i.e. $2, 4$ are of interest to us. Therefore we only need two values of $A(\mu \to \nu)$, which depend only on the temperature. These can by calculated once and for all in the initialization phase of the program, stored in an array and avoid the repeated calculation of the exponential $\mathrm{e}^{-\beta(E_\nu - E_\mu)}$ which is expensive.

In our program we also need to implement the calculation of the observables that we want to measure. These are the energy (13.13)

$$E = -\sum_{\langle ij \rangle} s_i s_j \,, \tag{13.24}$$

and the magnetization

$$M = \left| \sum_i s_i \right| \,. \tag{13.25}$$

Beware of the absolute value in the last equation! The Hamiltonian $H$ has a $Z_2$ symmetry because it is symmetric under reflection of all the spins. The probability of appearance of a state depends only on the value of $H$, therefore two configurations with opposite spin are equally probable. But such configurations have opposite magnetization, therefore the average magnetization $\langle \sum_i s_i \rangle$ will be zero due to this cancellation[18].

We can measure the energy and the magnetization in two ways. The first one is by updating their values each time a Metropolis step is accepted. This is easy and cheap since the difference in the sum in equations (13.24) and (13.25) depends only on the value of the spin $s_k^\mu$ and its nearest neighbors. The energy difference is already calculated by (13.23) whereas the difference in the magnetization in (13.25) is given by

$$\sum_i s_i^\nu - \sum_i s_i^\mu = s_k^\nu - s_k^\mu = -2 s_k^\mu \tag{13.26}$$

The second way is by calculating the full sums in (13.24) and (13.25) every time that we want to take a measurement. The optimal choice depends on how often one obtains a statistically *independent* measurement[19]. If the average acceptance ratio is $\bar{A}$, then the calculation of the magnetization using the first method requires $\bar{A}N$ additions per $N$ Monte Carlo steps, whereas the second one requires $N$ additions per measurement.

[18] This does not show for very small temperatures in the simulation with the Metropolis algorithm. As we decrease the temperature $\beta \gg \beta_c$, it takes many improbable steps to move from a state with $\sum_i s_i = M_1$ to a state with $\sum_i s_i = -M_1$. The Monte Carlo simulation consists of a finite number of steps, therefore we may obtain a non zero $\langle \sum_i s_i \rangle$, an incorrect result.

[19] This is given by the autocorrelation time, which will be discussed in detail later.

We use the normalization

$$\langle e \rangle = \frac{1}{N_l}\langle E \rangle = \frac{1}{2N}\langle E \rangle \,, \tag{13.27}$$

which gives the energy per link. We have that $-1 \leq e \leq +1$, where $e = -1$ for the ground state in which all $2N$ links have energy equal to $-1$. The magnetization per site is

$$\langle m \rangle = \frac{1}{N}\langle M \rangle \,. \tag{13.28}$$

We have that $0 \leq m \leq 1$, where $m = 0$ for $\beta = 0$ (perfect disorder) and $m = 1$ for the ground state at $\beta = \infty$ (perfect order). We call $m$ the order parameter since its value determines the phase that the system is in.

The specific heat is given by the fluctuations of the energy

$$c = \beta^2 N \langle (e - \langle e \rangle)^2 \rangle = \beta^2 N (\langle e^2 \rangle - \langle e \rangle^2) \,, \tag{13.29}$$

and the magnetic susceptibility by the fluctuations of the magnetization

$$\chi = \beta N \langle (m - \langle m \rangle)^2 \rangle = \beta N (\langle m^2 \rangle - \langle m \rangle^2) \,. \tag{13.30}$$

In order to estimate the amount of data necessary for an accurate measurement of these quantities, we consider the fact that for $n$ independent measurements the statistical error drops as $\sim 1/\sqrt{n}$. The problem of determining how often we have independent measurements is very important and it will be discussed in detail later in this chapter.

### 13.3.1 The Program

In this section we discuss the program[20] that implements the Monte Carlo simulation of the Ising model. The code in this section can be found in the accompanying software of this chapter in the directory `Ising_Introduction`.

In the design of the code, we follow the philosophy of modular programming. Different independent sections of the program will be coded in different files. This makes easier the development, maintenance and correction of the code by one or a team of programmers. A *header file* contains the definitions which are common for the code in one or more files. Then, all the parameters and common blocks are in one place and

[20]The basic ideas in the program are taken from the book by Newmann and Barkema [4].

they are easier to modify for all program units in a unified way, therefore avoiding errors. In our case we have only one such file, named `include.inc`, whose code will be included in the beginning of each program unit using an `include` statement:

```
!============== include.inc ==================
 implicit none
 integer,parameter        :: L = 12
 integer,parameter        :: N = L*L
 integer,parameter        :: XNN = 1, YNN = L
 integer,dimension(N)     :: s
 real(8),dimension(0:4)   :: prob
 real(8)                  :: beta
 common /lattice/            s
 common /parameters/         beta,prob
!function definitions:
 real(8)                  :: drandom
 integer                  :: seed
 common /randoms/            seed
```

The lattice size $L$ is a constant parameter, whereas the arrays and variables encoding the spins and the simulation parameters are put in common blocks. The array `s(N)` stores the spin of each lattice site which take values $\pm 1$. The variable `beta` is the temperature $\beta$ and the array `prob(0:4)` stores the useful values of the acceptance ratios $A(\mu \to \nu)$ according to the discussion on page 508. The function `drandom()` is the one discussed in section 11.1, which generates pseudorandom numbers uniformly distributed in the interval $(0, 1)$ - 0 and 1 excluded. The parameters `XNN` and `YNN` are used for computing the nearest neighbors in the `X` and `Y` directions according to the discussion of section 13.3 on helical boundary conditions. For example, for an internal site `i`, `i+XNN` is the nearest neighbor in the $+x$ direction and `i-YNN` is the nearest neighbor in the $-y$ direction.

The main program is in the file `main.f90` and drives the simulation:

```
!============== main.f90     ==================
program Ising2D
 include 'include.inc'
 integer :: start      !start= 0 (cold)/1 (hot)
 integer :: isweep, nsweep

 nsweep = 1000
 beta=0.21D0;seed=9873;start=1;
 call init(start)
```

```
 do isweep = 1, nsweep
  call met
  call measure
 end do
end program Ising2D
```

In the beginning we set the simulation parameters. The initial configuration is determined by the value of `start`. If `start=0`, then it is a cold configuration and if `start=1`, then it is a hot configuration. The temperature is set by the value of `beta` and the number of sweeps of the lattice by the value of `nsweep`. One sweep of the lattice is defined by `N` attempted spin flips. The flow of the simulation is determined by the initial call to `init`, which performs all initialization tasks, and the subsequent calls to `met` and `measure`, which perform `nsweep` Metropolis sweeps and measurements respectively.

One level down lies the subroutine `init`. The value of `start` is passed through its argument so that the desired initial state is set:

```
!============== init.f90     ==================
! file init.f90
! init(start): start = 0: cold start
!              start = 1: hot  start
!=============================================
subroutine init(start)
 include 'include.inc'
 integer :: start
 integer :: i
!----------------------
!initialize probabilities for E_\nu > E_mu
 prob=0.0D0
 do i=2,4,2 !i = dE/2 = (E_nu-E_mu)/2=2,4
  prob(i) = exp(-2.0D0*beta*i)
 enddo
!initial configuration:
 select case(start)
  case(0)!cold:
   s = 1 !all s(i) = 1
  case(1)!hot:
   do i=1,N
    if(drandom() .lt. 0.5D0)then
     s(i) =  1
    else
     s(i) = -1
    endif
   enddo
  case default
```

```
    print *,'init: start= ',start,' not valid. Exiting...'
    stop
  end select

end subroutine init
```

At first the array `prob(0:4)` is initialized to the values of the acceptance ratios $A(\mu \to \nu) = \mathrm{e}^{-\beta(E_\nu - E_\mu)} = \mathrm{e}^{-2\beta s_k^\mu \left(\sum_{\langle ik \rangle} s_i^\mu\right)}$. Those probabilities are going to be used when $s_k^\mu \left(\sum_{\langle ik \rangle} s_i^\mu\right) > 0$ and the possible values are obtained when this expression takes the values 2 and 4. These are the values stored in the array `prob(0:4)`, and we remember that the index of the array is the expression $s_k^\mu \left(\sum_{\langle ik \rangle} s_i^\mu\right)$, when it is positive.

The initial spin configuration is determined by the integer `start`. Using the `select case` block allows us to add more options in the future. When `start=0` all spins are set equal to 1, whereas when `start=1` each spin's value is set to $\pm 1$ with equal probability. The probability that `drandom()<0.5` is[21] $1/2$ in which case we set `s(i)=1`, otherwise (probability $1 - 1/2 = 1/2$) we set `s(i)=-1`.

The heart of the program is the subroutine `met()` which attempts `N` Metropolis steps. It picks a random site `N` times and asks the question whether to perform a spin flip. This is done using the Metropolis algorithm by calculating the change in the energy of the system before and after the change of the spin value according to (13.23):

```
!============== met.f90      ==================
subroutine met()
 include 'include.inc'
 integer :: i,k
 integer :: nn,snn,dE

 do k=1,N
!pick a random site:
  i = INT(N*drandom())+1
!snn=sum of neighboring spins:
  snn = 0
  nn=i+XNN;if(nn.gt.N)nn=nn-N;snn = snn + s(nn)
  nn=i-XNN;if(nn.lt.1)nn=nn+N;snn = snn + s(nn)
  nn=i+YNN;if(nn.gt.N)nn=nn-N;snn = snn + s(nn)
  nn=i-YNN;if(nn.lt.1)nn=nn+N;snn = snn + s(nn)
!dE=change in energy/2:
  dE=snn*s(i)
!flip:
```

[21] Remember that for the uniform distribution, $P(x < a) = a$

```
  if(dE.le.0)then
   s(i) = -s(i) !accept
  else if(drandom() < prob(dE))then
   s(i) = -s(i) !accept
  endif
 enddo !do k=1,N: end sweep
end subroutine met
```

The line

```
  i = INT(N*drandom())+1
```

picks a site `i=1,...,N` with equal probability. It is important that the value `i=N+1` never appears, something that happens if `drandom()=1.0`. This value has been excluded according to the discussion in section 11.1.

Next, we calculate the sum $\left(\sum_{\langle ik\rangle} s_i^\mu\right)$ in (13.23). The nearest neighbors of the site `i` have to be determined and this happens in the lines

```
  snn = 0
  nn=i+XNN;if(nn.gt.N)nn=nn-N;snn = snn + s(nn)
  nn=i-XNN;if(nn.lt.1)nn=nn+N;snn = snn + s(nn)
  nn=i+YNN;if(nn.gt.N)nn=nn-N;snn = snn + s(nn)
  nn=i-YNN;if(nn.lt.1)nn=nn+N;snn = snn + s(nn)
```

The variable `delta` is set equal to the product (13.23) $s_k^\mu\left(\sum_{\langle ik\rangle} s_i^\mu\right)$. If it turns out to be negative, then the change in energy is negative and the spin flip is accepted. If it turns out to be positive, then we apply the criterion set by (13.22) by using the array `prob(delta)`, which has been set in the subroutine `init`. The probability that `drandom()<prob(delta)` is equal to `prob(delta)`, in which case the spin flip is accepted. In all other cases, the spin flip is rejected and `s(i)` remains the same.

After each Metropolis sweep we perform a measurement. The code is minimal and simply prints the value of the energy and the magnetization to the `stdout`. The analysis is assumed to be performed by external programs. This way we keep the production code simple and store the raw data for a detailed and flexible analysis. The printed values of the energy and the magnetization will be used as monitors of the progress of the simulation, check thermalization and measure autocorrelation times. Plots of the measured values of an observable as a function of the Monte Carlo "time" are the so called "time histories". Time histories of appropriately chosen observables should *always* be viewed and used in order to check the progress and spot possible problems in the simulation.

The subroutine `measure` calculates the total energy and magnetization (without the absolute value) by a call to the functions `E()` and `M()`, which apply the formulas (13.24) and (13.25).

```
!============== measure.f90 ==================
subroutine measure()
 include 'include.inc'
 integer :: E,M
 print *, E(),M()
end subroutine measure
!=====================
integer function E()
 include 'include.inc'
 integer en,sum,i,nn
 en = 0
 do i=1,N
!Sum of neighboring spins: only forward nn necessary in the sum
  sum = 0
  nn=i+XNN;if(nn.gt.N)nn=nn-N;sum = sum + s(nn)
  nn=i+YNN;if(nn.gt.N)nn=nn-N;sum = sum + s(nn)
  en=en+sum*s(i)
 enddo
 e = -en
end function E
!=====================
integer function M()
 include 'include.inc'
 M=SUM(s)
end function M
```

The compilation of the code is done with the command

```
> gfortran main.f90 met.f90 init.f90 measure.f90 drandom.f90 \
   -o is
```

which results in the executable file `is`:

```
> ./is > out.dat
> less out.dat
-52 10
-48 40
-64 44
-92 64
......
```

The output of the program is two columns with the values of the to-

tal energy and magnetization (without the absolute value). In order to construct their time histories we give the gnuplot commands:

```
gnuplot> plot "out.dat" using 1 with lines
gnuplot> plot "out.dat" using 2 with lines
gnuplot> plot "out.dat" using (($2>0)?$2:-$2) with lines
```

The last line calculates the absolute values of the second column. The C-like construct ($2>0)?$2:-$2 checks whether the expression ($2>0) is true. If it is, then it returns $2, otherwise it returns -$2.

### 13.3.2 Towards a Convenient User Interface

In this section we will improve the code, mostly at the user interface level. This is a nice exercise on the interaction of the programming language with the shell and the operating system. The code presented can be found in the accompanying software of this chapter in the directory Ising_Metropolis.

An annoying feature of the program discussed in the previous section is that the simulation parameters are hard coded and the user needs to recompile the program each time she changes them. This is not very convenient if she has to do a large number of simulations. Another notable change that needs to be made in the code is that the final configuration of the simulation must be saved in a file, in order to be read as an initial configuration by another simulation.

One of the parameters that the user might want to set interactively at run time is the size of the lattice L. But this parameter determines the required memory for the array s(N). Therefore we have to use dynamic memory allocation for this array using the intrinsic function ALLOCATE. Another problem is that the array s(N) needs to be accessible by several parts of the program and allocatable arrays cannot be put in a common block. Another mechanism for sharing data among different functions and subroutines is the use of *modules*. This is the preferable method of doing it in modern Fortran programs where the use of common blocks is discouraged. The shared data needs to be put between the following statements:

```
module global_data
 implicit none
 SAVE
 ....
```

```
end module global_data
```

In place of the ... we can put variable declarations. We use the statement `SAVE` so that their values are saved between function and subroutine calls. The module has a name which in our case is `global_data`. Each program unit that needs to have access to its data needs to start with the statement `use global_data`:

```
subroutine share_global_data
 use global_data
 implicit none
 ....
end subroutine share_global_data
```

In the file `global_data.f90` we put all the global variables as follows:

```
module global_data
 implicit none
 SAVE
 integer                    :: L
 integer                    :: N
 integer                    :: XNN, YNN
 integer,allocatable        :: s(:)
 real(8),dimension(0:4)     :: prob
 real(8)                    :: beta
 integer                    :: nsweep,start
 integer                    :: seed,ranlux_level
 real(8)                    :: acceptance
 character(1024)            :: prog
end module global_data
```

The array `s(:)` is allocatable and its storage space will be allocated in the subroutine `init`. The variables `L`, `N`, `XNN` and `YNN` are not parameters anymore and their values will also be set in `init`. The new variables are `acceptance` which computes the fraction of accepted spin flips in a simulation, `ranlux_level` which determines the luxury level of RANLUX and `prog` which stores the command line name of the program that runs the simulation.

The main program has very few changes:

```
!============== main.f90     ==================
program Ising2D
 use global_data
 implicit none
```

```
 integer :: isweep

 call init
 do isweep = 1, nsweep
  call met
  call measure
 end do
 call endsim
end program Ising2D
```

Notice the line `use global_data` which gives access to the data in the module `global_data`. This is the first line of all program units. The subroutine `endsim` finishes off the simulation. Its most important function is to store the final configuration to a file for later use.

The subroutine `init` is changed quite a bit since it performs most of the functions that have to do with the user interface:

```
!============== init.f90     ==================
! start = 0: cold start
! start = 1: hot  start
! start = 2: use old configuration
!=============================================
subroutine init
 use global_data
 implicit none
 integer                      :: i,chk
 real(8)                      :: obeta=-1.0D0,r
 integer                      :: OL=-1
 character(1024)              :: buf
 integer,parameter            :: f_in=17 !file unit
 integer                      :: seeds(25)
!_____________________
!Define parameters from options:
 L=-1;beta=-1.0D0;nsweep=-1;start=-1;seed=-1
 ranlux_level=3
 call get_the_options
 if(start.EQ.0 .OR. start.EQ.1)then
  if(L    < 0    )call locerr('L    has not been set.')
  if(seed < 0    )call locerr('seed has not been set.')
  if(beta < 0.0D0)call locerr('beta has not been set.')
!Derived parameters:
  N=L*L;XNN=1;YNN=L
!Allocate memory for the spins:
  ALLOCATE(s(N),STAT=chk)
  if(chk      > 0)call locerr('allocation failure for s(N)')
 endif !if(start.EQ.0 .OR. start.EQ.1)
 if(start     < 0)call locerr('start  has not been set.')
```

```
 if(nsweep    < 0)call locerr('nsweep has not been set.')
!-----------------------
!initialize probabilities for E_\nu > E_mu
 prob=0.0D0
 do i=2,4,2 !i = dE/2 = (E_nu-E_mu)/2=2,4
  prob(i) = exp(-2.0D0*beta*i)
 enddo
 acceptance = 0.0D0
!-----------------------------------------
!initial configuration: cold(0),hot(1),old(2)
!-----------------------------------------
 select case(start)
!-----------------------------------------
  case(0)!cold:
   call simmessage(6)
   call RLUXGO(ranlux_level,seed,0,0)
   s = 1 !all s(i) = 1
!-----------------------------------------
  case(1)!hot:
  call simmessage(6)
   call RLUXGO(ranlux_level,seed,0,0)
   do i=1,N
    call ranlux(r,1)
    if(r .lt. 0.5D0)then
     s(i) =  1
    else
     s(i) = -1
    endif
   enddo
!-----------------------------------------
  case (2)!old:
   open(f_in,file='conf',status='OLD',ERR=101)
   read(f_in,*)buf !read in a comment line
   read(f_in,'(A4,I5,A4,I5,A6,G28.17,A6,25I16)')&
    buf,OL,buf,OL,buf,beta,buf,seeds
   if(L  < 0 ) L = OL !if L has not been set, read from file
   if(L /= OL) &  ! /= the same as .NE. (not equal)
    call locerr('L different from the one read from conf.')
   N=L*L;XNN=1;YNN=L
   !option's beta overrides conf's
   if(beta < 0.0D0) beta = obeta
!Allocate memory for the spins:
   ALLOCATE(s(N),STAT=chk);
   if(chk      > 0)call locerr('allocation failure for s(N)')
   call simmessage(6)
   print '(A)','# Reading configuration from file conf'
   do i=1,N
    read(f_in,*,END=102) s(i)
    if(s(i) /= 1 .AND. s(i) /= -1)&
```

```
      call locerr('wrong value of spin')
    enddo
    close(f_in)
    if(seed < 0) then !initialize from seeds read from file:
     call RLUXIN(seeds)
    else              !option seed sets new seed:
     call RLUXGO(ranlux_level,seed,0,0)
    endif
!-----------------------------------------------
   case default
    print *,'init: start= ',start,' not valid. Exiting...'
    stop 1
   end select
!-----------------------------------------------
   return
!here we put error messages:
101 call locerr('Configuration file conf not found.')
102 call locerr('File conf ended before reading all spins.')
end subroutine init
```

In the beginning, the simulation parameters that are to be determined by the user are given invalid default values. This way they are flagged as not been set. The subroutine[22] `get_the_options` sets the parameters to the values that the user passes through the command line:

```
 L=-1;beta=-1.0D0;nsweep=-1;start=-1;seed=-1
 call get_the_options
```

Upon return of `get_the_options`, one has to check if all the parameters have been set to acceptable values. For example, if the user has forgotten to set the lattice size `L`, the call to the subroutine `locerr` stops the program and prints the error message passed through its argument:

```
  if(L < 0 )call locerr('L has not been set.')
```

When the value of `N` is calculated from `L`, the program allocates memory for the array `s(N)`:

```
  N=L*L
  ALLOCATE(s(N),STAT=chk)
  if(chk > 0)call locerr('allocation failure for s(N)')
```

[22]Found in the file `options.f90`.

When memory allocation is successful, the variable `chk` is set to 0 by `ALLOCATE`. Otherwise we stop the program with a call to `locerr`.

Using the construct `SELECT CASE(start)` we set the initial configuration of the simulation. A value of `start=0` sets all spins equal to 1. The subroutine `simmmessage(f_unit)` prints important information about the simulation to the unit `f_unit`. The random number generator `RANLUX` is initialized with a call to `RLUXGO` according to the discussion in section 11.2, page 461. The global variable `ranlux_level` is set to 3 by default, but the user can change it from the command line (see `get_the_options`). If `start=0` the initial configuration is hot.

If `start=2` we attempt to read a configuration stored in a file named `conf`. The format of the file is strictly set by the way we print the configuration in the subroutine `endsim`. If the file does not exist, the argument `ERR=101` transfers the control of the program to the labeled statement with label 101. This is near the end of the program and stops the program with a call to `locerr`. In order to read the configuration properly we need to know the format of the data in the file `conf` which is, more or less, as follows:

```
# Configuration of 2d Ising model on square lattice....
Lx= 12 Ly= 12 beta= 0.21 seed=  3718479 5267541 12092770 ....
 -1
  1
  1
  1
 -1
.....
```

All comments of the first line are discarded in the character variable `buf`. The parameters `L` and `beta` of the stored configuration are stored in temporary variables `OL, obeta`, so that they can be compared with the values set by the user. This way, if the user has not set the temperature, the old configuration's temperature will be used, otherwise the user's value of the temperature will be used.

The same is true for seeding `RANLUX`: If the user provides a seed, then her seed will be used for seeding. Otherwise `RANLUX` is initialized to the state read from the file `conf`. Both choices are desirable in different cases: If the user wants to split a long simulation into several short runs, then each time she wants to restart the random number generator at exactly the same state. If she wants to use the same configuration in order to produce many *independent* results, then `RANLUX` has to produce different

sequences of random numbers each time[23]. This feature is coded in the lines:

```
   if(seed < 0) then !initialize from seeds read from file:
    call RLUXIN(seeds)
   else              !option seed sets new seed:
    call RLUXGO(ranlux_level,seed,0,0)
   endif
```

When reading the spins, we have to make sure that they take only the legal values $\pm 1$ and that the data is enough to fill the array `s(N)`[24]. Reading enough data is checked by the `READ` argument `END=102`. If the `READ` statement attempts to read past the end of the file `conf`, then the control of the program is transferred to the labeled statement with label 102. This will happen, e.g. if we attempt to read from a corrupted file.

The subroutine `endsim` saves the last configuration in the file `conf` and can be found in the file `end.f90`:

```
!============== end.f90      ==================
subroutine endsim()
 use global_data
 implicit none
 integer,parameter :: f_out = 17
 integer           :: i,seeds(25)

 call RLUXUT(seeds)
 call rename('conf','conf.old')
 open(unit=f_out,file='conf')
 write(f_out,'(A)')&
  '# Configuration of 2d Ising model on square lattice...'
 write(f_out,'(A4,I5,A4,I5,A6,G28.17,A6,25I16)')&
  'Lx= ',L,' Ly= ',L,' beta= ',beta,' seed= ',seeds
 do i=1,N
  write(f_out,'(I3)')s(i)
 enddo
 close(f_out)
 print '(A,F7.3)','# acceptance= ',&
  acceptance/DBLE(N)/DBLE(nsweep)
end subroutine endsim
```

[23]Assuming that the configuration in `conf` is thermalized, the simulations become statistically independent after time $2\tau$, where $\tau$ is the autocorrelation time.

[24]Note that we use the equivalent comparison operators `'>'`⇔`'.GT.'`, `'>='`⇔`'.GE.'`, `'/='`⇔`'.NE.'`, `'=='`⇔`'.EQ.'` etc.

The state of the random number generator RANLUX is saved by a call to RLUXUT which stores the necessary information in the array seeds. The call to the subroutine RENAME renames the file conf (if it exists) to the backup file conf.old. The format (A4, I5, A4, I5, A6, G28.17, A6, 25I16) has to be obeyed strictly during the output, as well as during the input of the configuration in the subroutine init.

The subroutine get_the_options() reads the parameters, passed through *options* from the command line. The choice to use options for passing parameters to the program has the advantage that they can be passed optionally and in any order desired. Let's see how they work. Assume that the executable file is named is. The command

```
> ./is -L 10 -b 0.44 -s 1 -S 5342 -n 1000
```

will run the program after setting L=10 (-L 10), beta=0.44 (-b 0.44), start=1 (-s 1), seed=5342 (-S 5342) and nsweep=1000 (-n 1000). The -L, -b, -s, -S, -n are *options* or *switches* and can be put in any order in the arguments of the command line. The arguments following an option are the values passed to the corresponding variables. Options can also be used without arguments, in which case a common use is to make the command function differently[25]. In our case, the option -h is an option without an argument which makes the program to print a usage message and exit without running the simulation:

```
> ./is -h
Usage: ./is [options]
       -L: Lattice length (N=L*L)
       -b: beta
       -s: start (0 cold, 1 hot, 2 old config.)
       -S: seed
       -n: number of sweeps and measurements
       -u: seed  from /dev/urandom
       -r: ranlux_level
Monte Carlo simulation of 2d Ising Model. Metropolis is used by
default. Using the options, the parameters of the simulations
must be set for a new run (start=0,1). If start=2, a
configuration is read from the file conf.
```

This is a way to provide a short documentation on the usage of a program.

Let's see the code, which is found in the file options.f90:

[25]Remember how the option -l changes the results of the command ls if executed as ls -l

```
!============== options.f90 ==================
subroutine get_the_options
 use global_data
 use getopt_m    !from getopt.f90
 implicit none
 call getarg(0,prog)

 do
  select case( getopt( "-hL:b:s:S:n:r:u" ))
  case( 'L' )
   read(optarg,*)L
  case( 'b' )
   read(optarg,*)beta
  case( 's' )
   read(optarg,*)start
  case( 'S' )
   read(optarg,*)seed
  case( 'n' )
   read(optarg,*)nsweep
  case( 'r' )
   read(optarg,*)ranlux_level
  case( 'u' )
   open (28, file="/dev/urandom", &
    access="stream", form="unformatted")
   read (28) seed
   seed = ABS(seed)
   close(28)
  case( 'h' )
   call usage
  case( '?' )
   print *, 'unknown option    ', optopt
   stop
  case( char(0)) ! done with options
   exit
  case( '-' )    ! use -- to exit from options
   exit
  case default
   print *, 'unhandled option ', optopt
  end select
 enddo
end subroutine get_the_options
```

The command `call getarg(0,prog)` stores the name of the program in the command line to the character variable `prog`. The function `getopt` is a function written by Mark Gates and its code is in the file `getopt.f90`. It is programmed so that its usage is similar to the corresponding `C` function[26].

[26]Read the comments in the file `getopt.f90` for more information.

The argument `"-hL:b:s:S:n:r:u"` in `getopt` defines the allowed options `'-'`, `'L'`, `'b'`, `'s'`, `'S'`, `'n'`, `'r'`, `'u'`. When a user passes one of those through the command line (e.g. `-L 100`, `-h`) the `do` loop takes us to the corresponding `CASE`. If an option does not take an argument (e.g. `-h`), then a set of commands can be executed, like `call usage`. If an option takes an argument, this is marked by a semicolon in the argument of `getopt` (e.g. `L:`, `b:`, ...) and the argument can be accessed through the `character` variable `optarg`. For example, the statements

```
  case( 'L' )
   read(optarg,*)L
```

and the command line arguments `-L 10` set `optarg` to be equal to `'10'`. Be careful, `'10'` is not a number, but a string of characters! In order to convert the *character* `'10'` to the *integer* 10 we use the command `READ`, where instead of a unit number in its arguments we put the variable name. We do the same for the other simulation parameters.

The subroutine `locerr` takes a character variable in its argument which prints it to the `stderr` together with the name of the program in the command line. Then it stops the execution of the program:

```
subroutine locerr(errmes)
 use global_data
 implicit none
 character(*) :: errmes
 write(0,'(A,A)'),TRIM(prog),':',TRIM(errmes),' Exiting....'
 stop 1
end subroutine locerr
```

Note the use of the intrinsic function `TRIM` which removes the trailing blanks of a character variable. If we hadn't been using it, the variable `character(1024) :: prog` would have been printed in 1024 character spaces, something that it wouldn't have been very pretty...

The subroutine `usage` is ... used very often! It is a constant reminder of the way that the program is used and helps users with weak long and/or short term memory!

```
subroutine usage
 use global_data
 implicit none
 print '(3A)','Usage: ',TRIM(prog),' [options]'
 print '( A)','        -L: Lattice length (N=L*L)'
```

```
 print '( A)','        -b: beta'
 print '( A)','        -s: start (0 cold, 1 hot, 2 old config.)'
 print '( A)','        -S: seed'
 print '( A)','        -n: number of sweeps and measurements'
 print '( A)','        -u: seed  from /dev/urandom'
 print '( A)','        -r: ranlux_level'
 print '( A)','Monte Carlo simulation of 2d Ising Model....'
 stop
end subroutine usage
```

The subroutine `simmessage` is also quite important. It "labels" our results by printing all the information that defines the simulation. It is very important to label all of our data with this information, otherwise it can be dangerously useless! Imagine a set of energy measurements without knowing the lattice size and/or the temperature... Other useful information may turn out to be crucial, even though we might not appreciate it at programming time: The name of the computer, the operating system, the user name, the date etc. By varying the unit number in the argument, we can print the same information in any file we want.

```
subroutine simmessage(unit)
 use global_data
 implicit none
 integer :: unit
 character(100) :: user,host,mach,tdate
 call GETLOG(user)
 call GETENV('HOST'     ,host)
 call GETENV('HOSTTYPE',mach)
 call FDATE (tdate)
 write(unit,'( A        )')&
  '# ##########################################################'
 write(unit,'( A        )')&
  '# 2d Ising Model, Metropolis algorithm on square lattice'
 write(unit,'( 8A       )')&
  '# Run on ',TRIM(host),' (',TRIM(mach),') by ',TRIM(user),&
  ' on ',TRIM(tdate)
 write(unit,'( A,I6,A   )')'# L       = ',L,' (N=L*L)'
 write(unit,'( A,I14    )')'# seed    = ',seed
 write(unit,'( A,I12,A )')'# nsweeps = ',nsweep,' (No. sweeps)'
 write(unit,'( A,G28.17)')'# beta    = ',beta
 write(unit,'( A,I4  ,A )')'# start   = ',start,&
  ' (0 cold, 1 hot, 2 old config)'
end subroutine simmessage
```

The compilation can be done with the command:

```
> gfortran global_data.f90 getopt.f90 \
  main.f90 init.f90 met.f90 measure.f90 end.f90 \
  options.f90 ranlux.F -o is
```

It is important to note that the files containing modules, like `global_data.f90` and `getopt.f90`, must precede the files with the code that use the modules.

In order to run the program we pass the parameters through options in the command line, like for example:

```
> /usr/bin/time ./is -L 10 -b 0.44 -s 1 -S 5342 -n 10000 \
    >& out.dat &
```

The command `time` is added in order to measure the computer resources (CPU time, memory, etc) that the program uses at run time.

A useful tool for complicated compilations is the utility `make`. Its documentation is several hundred pages which can be accessed through the `info` pages[27] and the interested reader is encouraged to browse through it. If in the current directory there is a file named `Makefile` whose contents[28] are

```
# ####################   Makefile   ############################
FC     =  gfortran
OBJS   =  global_data.o getopt.o ranlux.o \
          main.o init.o met.o measure.o end.o options.o
FFLAGS = -O2

is: $(OBJS)
    $(FC) $(FFLAGS) $^ -o $@

$(OBJS):   global_data.f90
options.o: getopt.f90
%.o: %.f90
    $(FC) $(FFLAGS)   -c -o $@ $<
```

then this instructs the program `make` how to "make" the executable file `is`. What have we gained? In order to see that, run `make` for the first time. Then try making a trivial change in the file `main.f90` and rerun

[27]Use the command `info make` or visit the www address `www.gnu.org/software/make/manual/make.html`

[28]Beware: one of the quirks of the program `make` is that all executable commands in `Makefile` *must* be in a line that starts exactly with a `TAB`. In the example `Makefile` shown above, the empty space before such lines is one `TAB` and *not* 8 empty spaces.

make. Then only the modified file is compiled and not the ones that have not been touched. This is accomplished by defining *dependencies* in Makefile which execute commands conditionally depending on the time stamps on the relevant files. Dependencies are defined in lines which are of the form keyword: word1 word2 .... For example, the line options.o: getopt.f90 defines a dependency of the file options.o from the file getopt.f90. Lines 2-4 in the above Makefile define variables which can be used in the commands that follow. There are *many* predefined variables[29] in make which makes make programming easier. By using make in a large project, we can automatically link to libraries, pass complicated compiler options, do conditional compilation (depending, e.g., on the operating system, the compiler used etc), etc. A serious programmer needs to invest some time in order to use the full potential of make for the needs of her project[30].

## 13.4 Thermalization

The problem of thermalization can be important for some systems studied with Monte Carlo simulations. Even though it will not be so important in the simulations performed in this book, we will discuss it because of its importance in other problems. The reader should bear in mind that the thermalization problem becomes more serious with increasing system size and when autocorrelation times are large.

In a Monte Carlo simulation, the system is first put in a properly chosen initial configuration in order to start the Markov process. In section 12.2 we saw that when a system is in thermal equilibrium with a reservoir at a given temperature, then a typical state has energy that differs very little from its average value and belongs to a quite restricted region of phase space. Therefore, if we choose an initial state that is far from this region, then the system has to perform a random walk in the space of states until it finds the region of typical states. This is the thermalization process in a Monte Carlo simulation.

There are two problems that need to be addressed: The first one is the appropriate choice of the initial configuration and the second one is to find criteria that will determine when the system is thermalized. For

[29] Try the command make -p

[30] At the time of the writing of this section, make had some problems with complicated Fortran compilations, which I hope they will be resolved in the future. You may also check out the program foray at code.google.com/p/foraytool or search the current "state of the art".

the Ising model the initial configuration is either, (a) cold, (b) hot *or* (c) old state. It is obvious that choosing a hot state in order to simulate the system at a cool temperature is not the best choice, and the system will take longer to thermalize than if we choose a cold state or an old state at a nearby temperature. This is clearly seen in figure 13.7. Thermalization

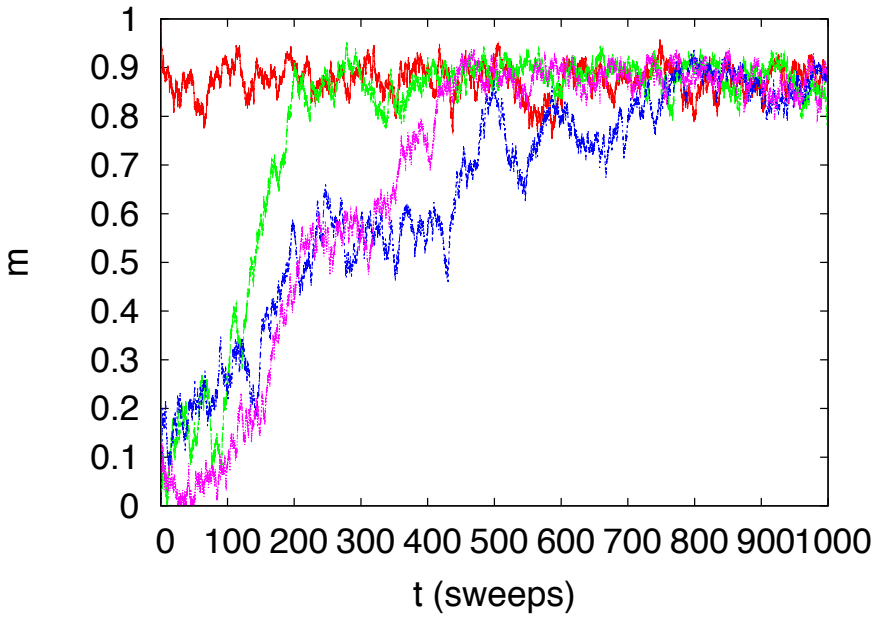



Figure 13.7: Magnetization per site for the Ising model in the ordered phase with $L = 40$, $\beta = 0.48$. We show the thermalization of the system by starting from a cold state and three hot ones. For a hot start, thermalization takes up to 1000 sweeps.

depends on the temperature and the system size, but it also depends on the physical quantity that we measure. Energy is thermalized faster than magnetization. In general, a local quantity thermalizes fast and a non local one slower. For the Ising model, thermalization is easier far from the critical temperature, provided that we choose an initial configuration in the same phase. It is easier to thermalize a small system rather than a large one.

The second problem is to determine when the system becomes thermalized and discard all measurements before that. One way is to start simulations using different initial states, or by keeping the same initial state and using a different sequence of random numbers. When the times histories of the monitored quantities converge, we are confident that the system has been thermalized. Figure 13.7 shows that the thermalization

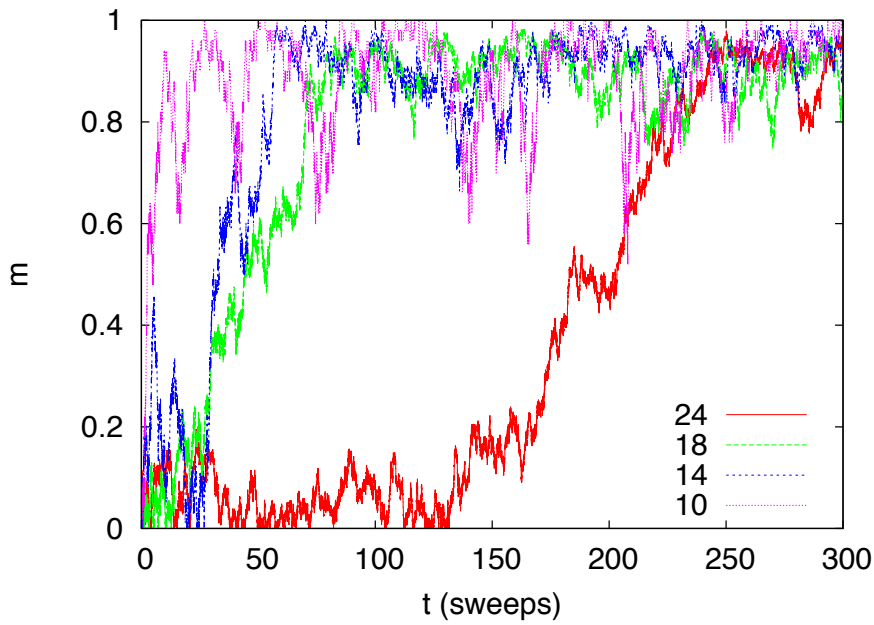



Figure 13.8: Magnetization per site for the Ising model for $L = 10, 14, 18, 24$ and $\beta = 0.50$. Thermalization from a hot start takes longer for a large system.

time can vary quite a lot.

A more systematic way is to compute an expectation value by removing an increasing number of initial measurements. When the results converge within the statistical error, then the physical quantity that we measure has thermalized. This process is shown in figures 13.10 and 13.11 where we progressively drop $0, 20, 50, 100, 200, 400, 800, 1600, 3200$ and $6400$ initial measurements until the expectation value of the magnetization stabilizes within the limits of its statistical error.

## 13.5 Autocorrelations

In order to construct a set of independent measurements using a Markov process, the states put in the sample should be statistically uncorrelated. But for a process using the Metropolis algorithm this is not possible. The next state differs from the previous one by at most one value of their spins. We would expect that we could obtain an almost statistically independent configuration after one spin update per site, a so called sweep of the lattice. This is indeed the case for the Ising model for temperatures far from the critical region. But as one approaches $\beta_c$, correlations between

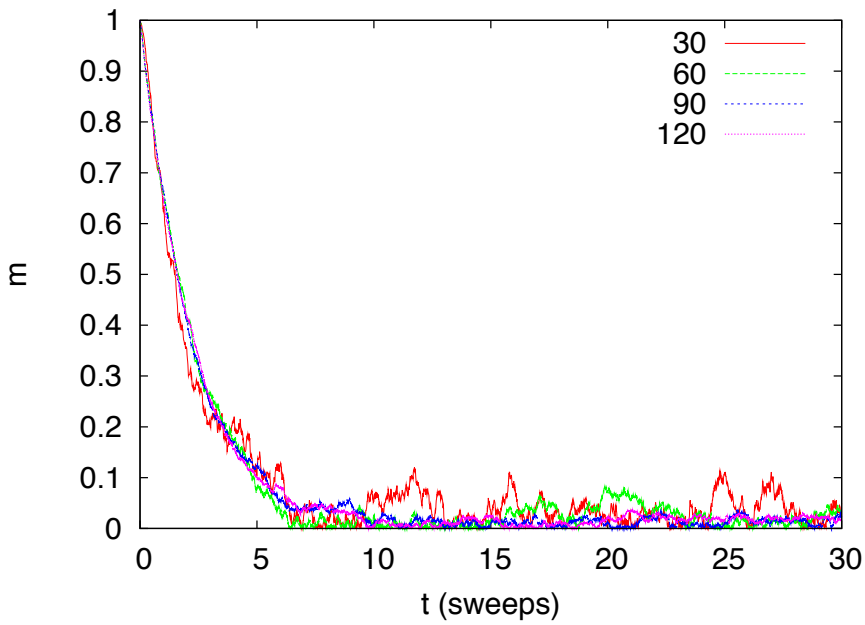



Figure 13.9: Magnetization per site for the Ising model for $L = 30, 60, 90, 120$ and $\beta = 0.20$. Thermalization starting from a cold start does not depend on the system size.

configurations obtained after a few sweeps remain strong. It is easy to understand why this is happening. As the correlation length $\xi$ (12.46) becomes much larger than a few lattice spacings, large clusters of same spins are formed, as can be seen in figure 13.36. For two statistically independent configurations, the size, shape and position of those clusters should be quite different. For a single flip algorithm, like the Metropolis algorithm, this process takes a lot of time[31].

For the quantitative study of autocorrelations between configurations we use the *autocorrelation function*. Consider a physical quantity $\mathcal{O}$ (e.g. energy, magnetization, etc) and let $\mathcal{O}(t)$ be its value after Monte Carlo "time" $t$. $t$ can be measured in sweeps or multiples of it. The autocorrelation function $\rho_{\mathcal{O}}(t)$ of $\mathcal{O}$ is

$$\rho_{\mathcal{O}}(t) = \frac{\langle(\mathcal{O}(t') - \langle\mathcal{O}\rangle)(\mathcal{O}(t'+t) - \langle\mathcal{O}\rangle)\rangle_{t'}}{\langle(\mathcal{O} - \langle\mathcal{O}\rangle)^2\rangle}, \tag{13.31}$$

where $\langle\ldots\rangle_{t'}$ is the average value over the configurations in the sample for $t' < t_{\max} - t$. The normalization is such that $\rho_{\mathcal{O}}(0) = 1$.

[31]The Metropolis algorithm changes the clusters mostly by modifying their boundaries, since it is less probable to change the value of a spin in the cluster where all its nearest neighbors have the same spin.

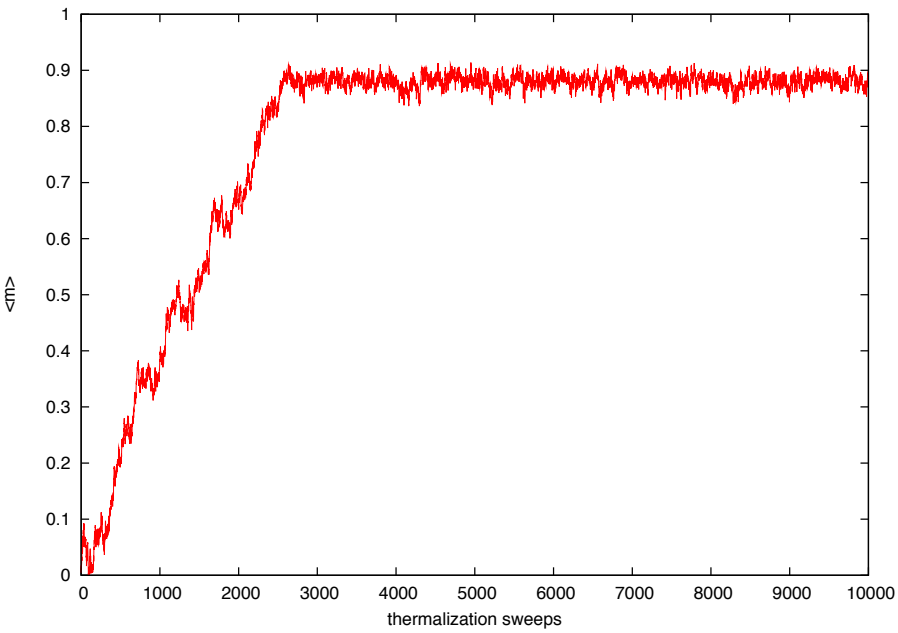



Figure 13.10: Magnetization per site for the Ising model with $L = 100$ and $\beta = 0.48$. Thermalization starts from a hot state.

The above definition reminds us the correlation function of spins in space (see equation (12.45)) and the discussion about its properties is similar to the one of section 12.4. In a few words, when the value of $\mathcal{O}$ after time $t$ is strongly correlated to the one at $t = 0$, then the product in the numerator in (13.31) will be positive most of the time and the value of $\rho_{\mathcal{O}}(t)$ will be positive. When the correlation is weak, the product will be positive and negative the same number of times and $\rho_{\mathcal{O}}(t)$ will be almost zero. In the case of anti-correlations $\rho_{\mathcal{O}}(t)$ is negative. Negative values of $\rho_{\mathcal{O}}(t)$ occur, but these are artifacts of the finite size of the sample and should be rejected.

Asymptotically $\rho_{\mathcal{O}}(t)$ drops exponentially

$$\rho_{\mathcal{O}}(t) \sim \mathrm{e}^{-t/\tau_{\mathcal{O}}} \ . \tag{13.32}$$

$\tau_{\mathcal{O}}$ is the time scale of decorrelation of the measurements of $\mathcal{O}$ and it is called the *autocorrelation time* of $\mathcal{O}$. After time $2\tau_{\mathcal{O}}$, $\rho_{\mathcal{O}}(t)$ has dropped to the $1/\mathrm{e}^2 \approx 14\%$ of its initial value and then we say that we have an independent measurement of[32] $\mathcal{O}$. Therefore, if we have $t_{\max}$ measurements,

[32]Autocorrelation times can be quite different for different $\mathcal{O}$.

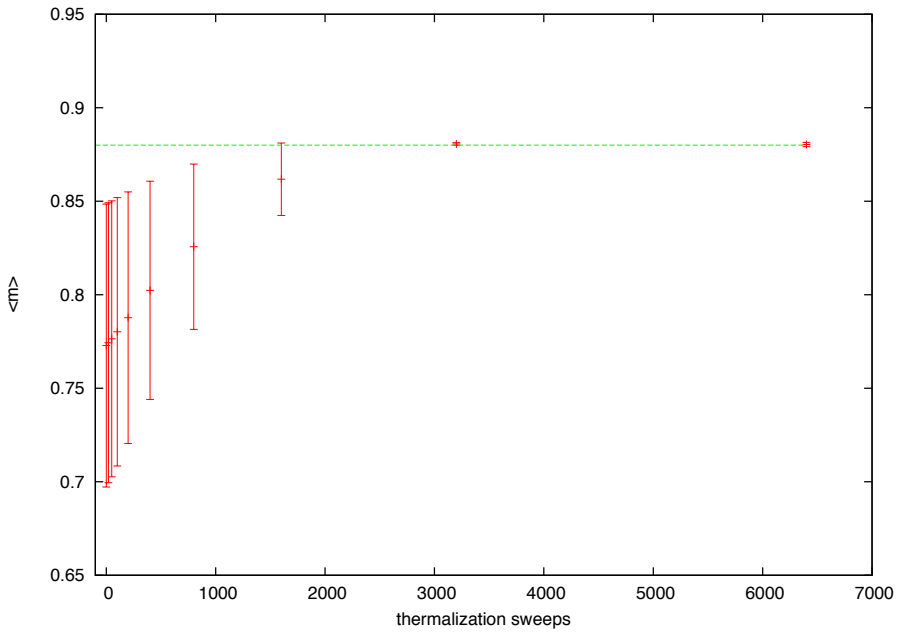



Figure 13.11: Magnetization per site for the Ising model with $L = 100$ and $\beta = 0.48$. We calculate the expectation value $\langle m \rangle$ by neglecting an increasing number of "thermalization sweeps" from the measurements in figure 13.10. When the neglected sweeps reach the thermalized state, the result converges to $\langle m \rangle = 0.880(1)$. This is an indication that the system has thermalized.

the number of independent measurements of $\mathcal{O}$ is

$$n_{\mathcal{O}} = \frac{t_{\max}}{2\tau_{\mathcal{O}}} \,. \tag{13.33}$$

For expensive measurements we should measure every $\sim \tau_{\mathcal{O}}$ sweeps. If the cost of measurement is not significant, then we usually measure more often, since there is still statistical information even in slightly correlated configurations. An accurate determination of $\tau_{\mathcal{O}}$ is not easy since it requires measuring for $t \gg \tau_{\mathcal{O}}$.

An example is shown in figure 13.12 for the case of the magnetization ($\mathcal{O} = m$). We calculate the function $\rho_m(t)$ and we see that a fit to equation (13.32) is quite good for $\tau_m = 235 \pm 3$ sweeps. The calculation is performed on a sample of $10^6$ measurements with 1 measurement/sweep. Therefore the number of independent measurements is $\approx 10^6/(2 \times 235) \approx 2128$.

Another estimator of the autocorrelation time is the so called *integrated autocorrelation time* $\tau_{\text{int},\mathcal{O}}$. Its definition stems from equation (13.32) where

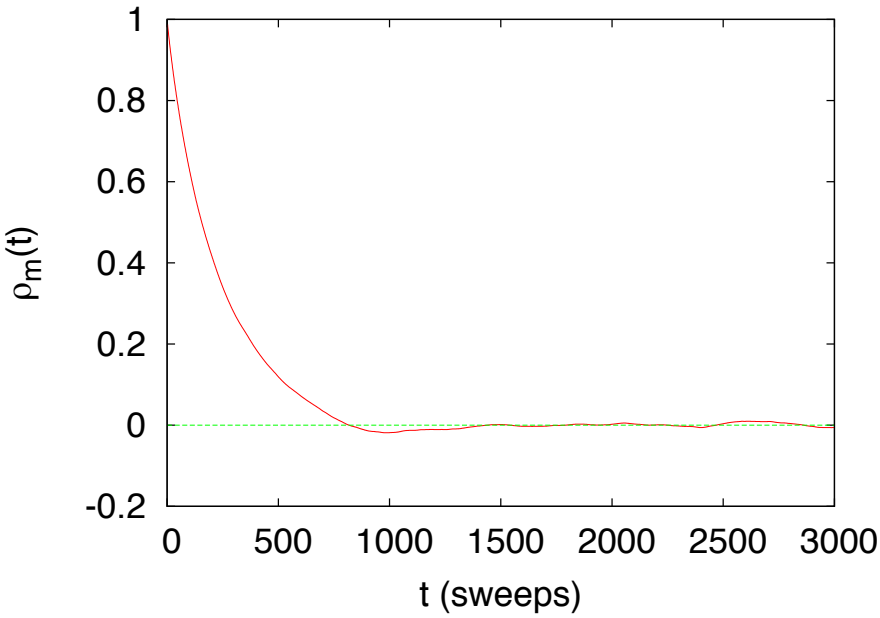



Figure 13.12: The autocorrelation function of the magnetization $\rho_m(t)$ for the Ising model for $L = 100$, $\beta = 0.42$. We see its exponential decay and that $\tau_m \approx 200$ sweeps. One can see the finite sample effects (the sample consists of about 1,000,000 measurements) when $\rho$ starts fluctuating around 0.

we take

$$\tau_{\text{int},\mathcal{O}} = \int_0^{+\infty} dt\, \rho_{\mathcal{O}}(t) \sim \int_0^{+\infty} dt\, \mathrm{e}^{-t/\tau_{\mathcal{O}}} = \tau_{\mathcal{O}} \,. \tag{13.34}$$

The values of $\tau_{\text{int},\mathcal{O}}$ and $\tau_{\mathcal{O}}$ differ slightly due to systematic errors that come from the corrections[33] to equation (13.32). The upper limit of the integral is cut off by a maximum value $t_{\text{max}}$

$$\tau_{\text{int},\mathcal{O}}(t_{\text{max}}) = \int_0^{t_{\text{max}}} dt\, \rho_{\mathcal{O}}(t) \,. \tag{13.35}$$

For large enough $t_{\text{max}}$ we observe a plateau in the plot of the value of $\tau_{\text{int},\mathcal{O}}(t_{\text{max}})$ which indicates convergence, and we take this as the estimator of $\tau_{\text{int},\mathcal{O}}$. For even larger $t_{\text{max}}$, finite sample effects enter in the sum that should be discarded.

[33] In our calculations, we will see differences of the order of 10%. The actual values can be different but their scaling properties are same.

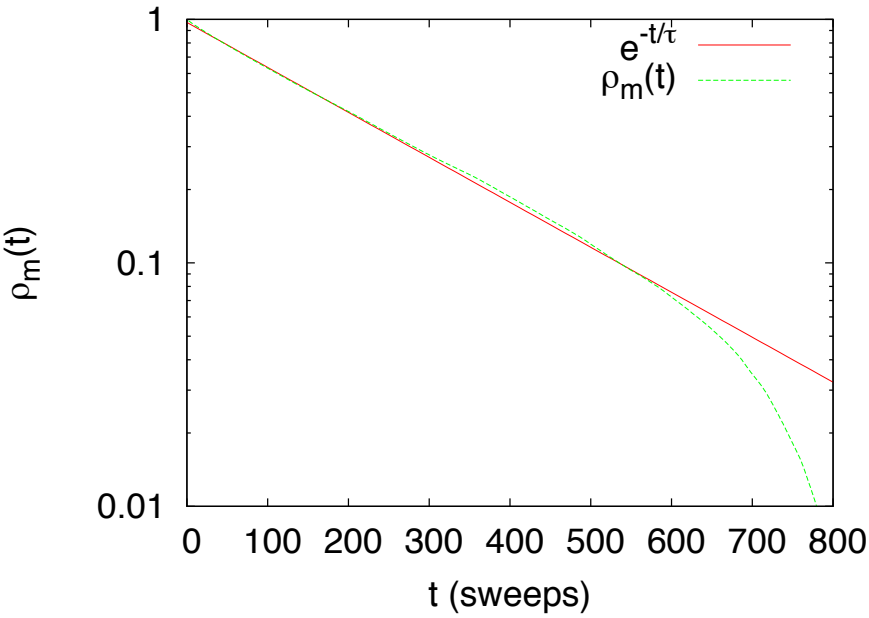



Figure 13.13: The autocorrelation function shown in figure 13.12 of the magnetization $\rho_m(t)$ for the Ising model for $L = 100$, $\beta = 0.42$ in a log plot. The plot shows a fit to $C\mathrm{e}^{-t/\tau}$ (see equation (13.32)) with $\tau = 235(3)$ sweeps.

This calculation is shown in figure 13.14 where we used the same measurements as the ones in figure 13.12. We find that $\tau_{\mathrm{int},m} = 217(3)$sweeps, which is somewhat smaller than the autocorrelation time that we calculated using the exponential fit to the autocorrelation function. If we are interested in the scaling properties of the autocorrelation time with the size of the system $L$ or the temperature $\beta$, then this difference is not important[34]. The calculation of $\tau_{\mathrm{int},\mathcal{O}}$ is quicker since it involves no fitting[35].

Autocorrelation times are not a serious problem away from the critical region. Figures 13.15 and 13.16 show that they are no longer than a few sweeps and that they are independent of the system size $L$. As

[34]The actual value of $\tau_{\mathcal{O}}$ is used in computing the number of independent configurations from (13.33) and the correction of the statistical error in (13.47). In both cases, the difference in the values of $\tau_{\mathcal{O}}$ is not significant. For (13.47), this is because the concept of the error of the error is slightly fuzzy.

[35]As we will see later, there are other, smaller autocorrelation times present as well. These are not taken into account in the definition of the integrated autocorrelation time and the detailed study of the autocorrelation function is necessary if more accuracy is desired.

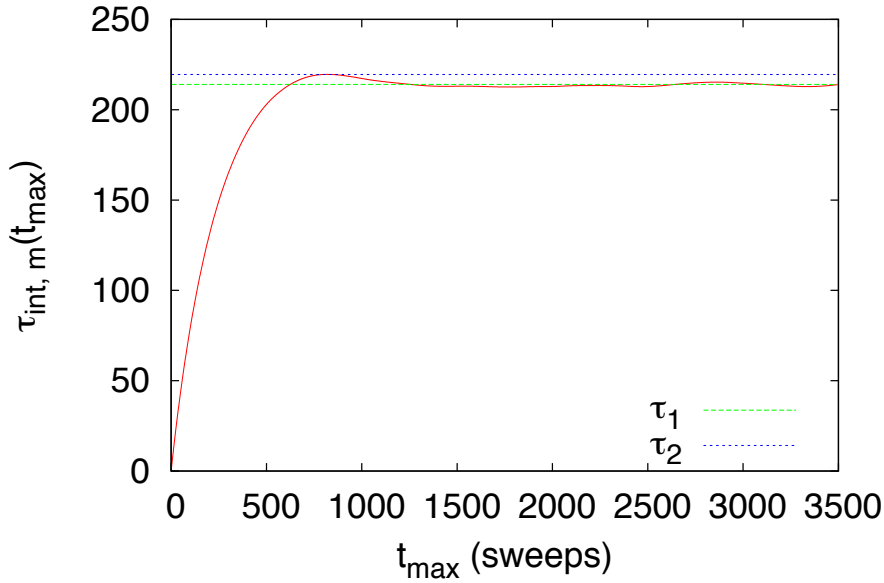



Figure 13.14: Calculation of the integrated autocorrelation time of the magnetization for the same data used in figure 13.19. There is a plateau in the values of $\tau_{\text{int},m}$ for $\tau_1 = 214(1)$sweeps and a maximum for $\tau_2 \approx 219.5$ sweeps. The fall from $\tau_2$ to $\approx \tau_1$ is due to the negative values of $\rho_m(t)$ due to the noise coming from finite sample effects. We estimate that $\tau_{\text{int},m} = 217(3)$ sweeps.

we approach the critical region, autocorrelation times increase. At the critical region we observe scaling of their values with the system size, which means that for large $L$ we have that

$$\tau \sim L^z \,. \tag{13.36}$$

This is the phenomenon of *critical slowing down*. For the Metropolis algorithm and the autocorrelation time of the magnetization, we have that $z = 2.1665 \pm 0.0012$ [57]. This is a large value and that makes the algorithm expensive for the study of the critical properties of the Ising model. It means that the simulation time necessary for obtaining a given number of independent configurations increases as

$$t_{\text{CPU}} \sim L^{d+z} \approx L^{4.17} \,. \tag{13.37}$$

In the next chapter, we will discuss the scaling relation (13.36) in more detail and present new algorithms that reduce critical slowing down drastically.

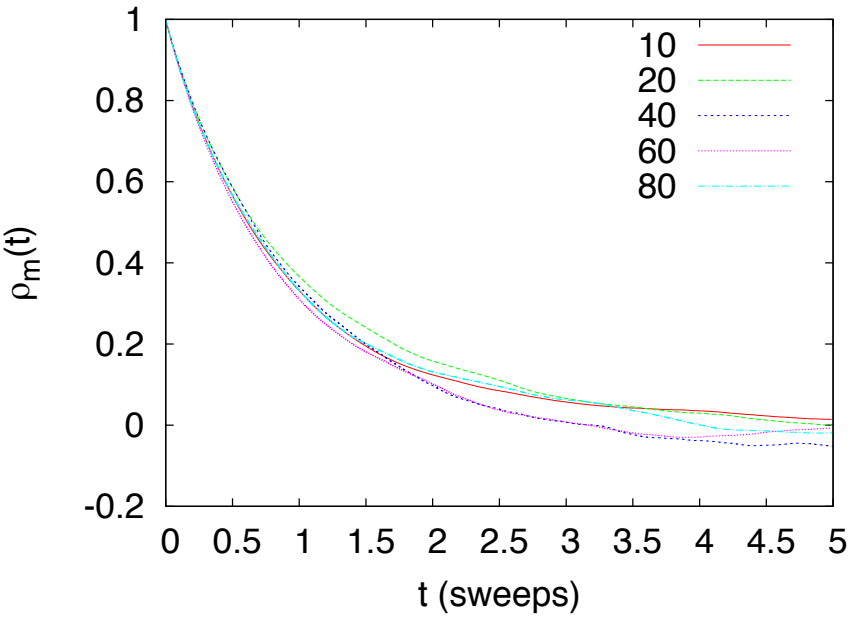



Figure 13.15: The autocorrelation time of the magnetization for the Ising model at (high) temperature $\beta = 0.20$ for $L = 10, 20, 40, 60, 80$. The autocorrelation time in sweeps is independent of $L$.

## 13.6 Statistical Errors

The estimate of the expectation value of an observable from its average value in a sample gives no information about the quality of the measurement. The complete information is provided by the full distribution, but in practice we are usually content with the determination of the "statistical error" of the measurement. This is defined using the assumption that the distribution of the measurements is Gaussian, which is a very good approximation if the measurements are *independent*. The statistical error is determined by the fluctuations of the values of the observable in the sample around its average (see discussion in section 12.2 and in particular equation (12.27)). Statistical errors can be made to vanish, because they decrease as the inverse *square root* of the size of the sample.

Besides statistical errors, one has *systematic* errors, which are harder to control. Some of them are easier to control (like e.g. poor thermalization) and others maybe hard even to realize their effect (like e.g. a subtle problem in a random number generator). In the case of a discrete, finite, lattice, approximating a continuous theory, there are systematic errors due to the discretization and the finite size of the system. These

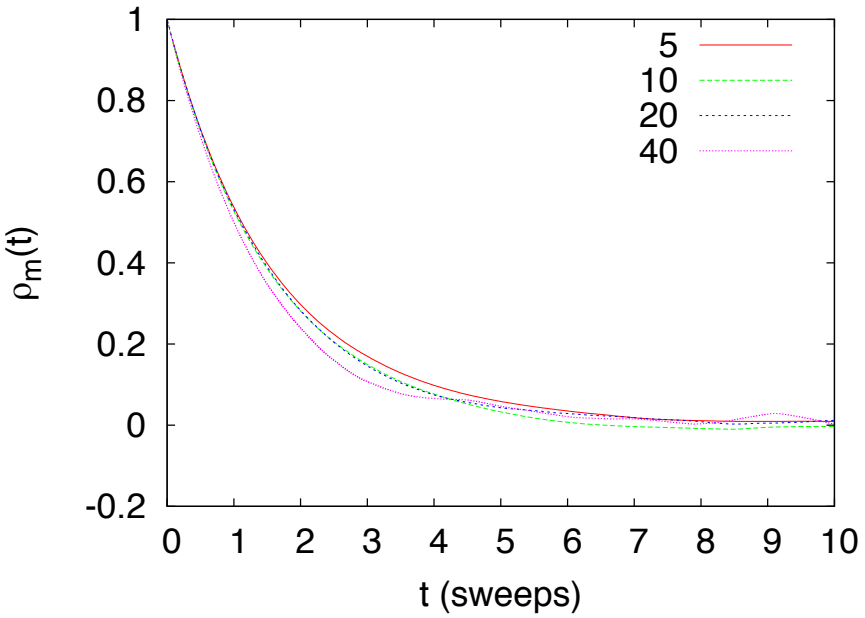



Figure 13.16: The autocorrelation time of the magnetization for the Ising model at (low) temperature $\beta = 0.65$ for $L = 5, 10, 20, 40$. The autocorrelation time in sweeps is independent of $L$.

errors are reduced by simulating larger systems and by using several techniques (e.g. finite size scaling) in order to extrapolate the results to the thermodynamic limit. These will be studied in detail in the following chapter.

### 13.6.1 Errors of Independent Measurements

Using the assumption that the source of statistical errors are the thermal fluctuations around the average value of an observable, we conclude that its expectation value can be estimated by the mean of the sample and its error by the error of the mean. Therefore if we have a sample of $n$ measurements $\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{n-1}$, their mean is an estimator of $\langle \mathcal{O} \rangle$

$$\langle \mathcal{O} \rangle = \frac{1}{n} \sum_{i=0}^{n-1} \mathcal{O}_i \,. \tag{13.38}$$

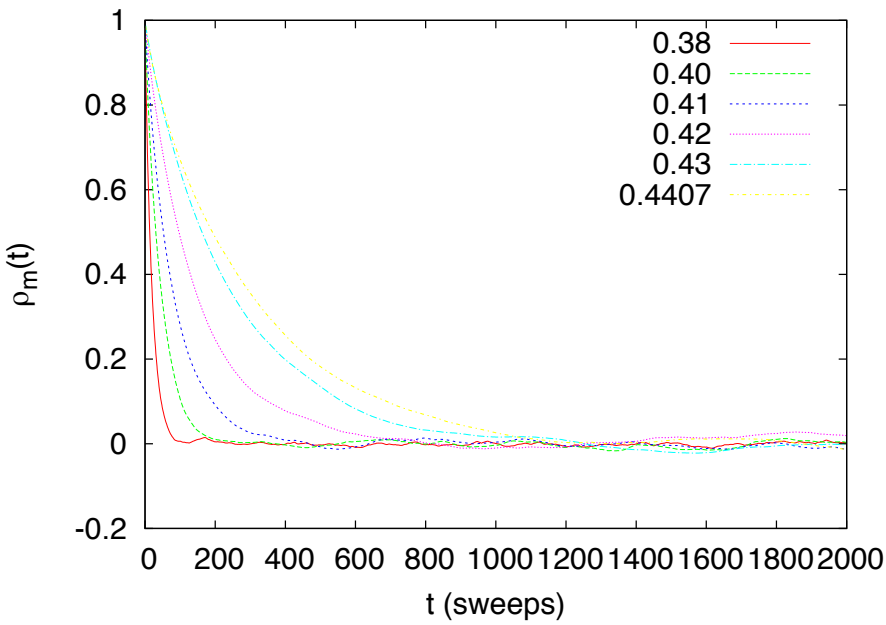



Figure 13.17: The autocorrelation function of the magnetization for the Ising model for $L = 40$. It shows how the autocorrelation time increases as we approach the critical temperature from the disordered (hot) phase.

The error of the mean is an estimator of the statistical error $\delta\mathcal{O}$

$$(\delta\mathcal{O})^2 \equiv \sigma_{\mathcal{O}}^2 = \frac{1}{n-1}\left\{\frac{1}{n}\sum_{i=0}^{n-1}(\mathcal{O}_i - \langle\mathcal{O}\rangle)^2\right\} = \frac{1}{n-1}\left(\langle\mathcal{O}^2\rangle - \langle\mathcal{O}\rangle^2\right) . \tag{13.39}$$

The above equations assume that the sample is a set of statistically independent measurements. This is not true in a Monte Carlo simulation due to the presence of autocorrelations. If the autocorrelation time, measured in number of measurements, is $\tau_{\mathcal{O}}$, then according to equation 13.33 we will have $n_{\mathcal{O}} = n/(2\tau_{\mathcal{O}})$ independent measurements. One can show that in this case, the statistical error in the measurement of $\mathcal{O}$ is[36] [58]

$$(\delta\mathcal{O})^2 = \frac{1+2\tau_{\mathcal{O}}}{n-1}\left(\langle\mathcal{O}^2\rangle - \langle\mathcal{O}\rangle^2\right) . \tag{13.40}$$

[36]See also chapter 4.1 in [5]

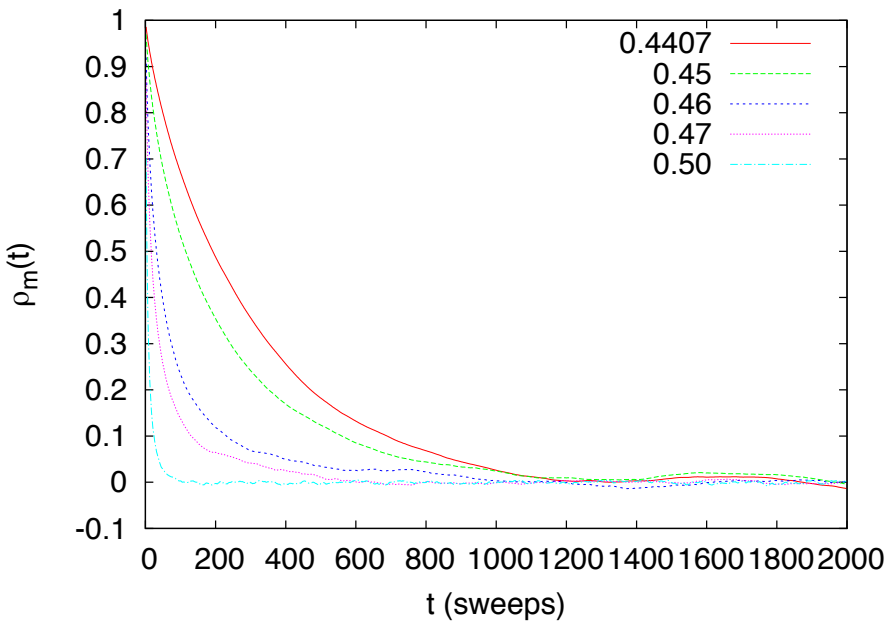



Figure 13.18: The autocorrelation function of the magnetization for the Ising model for $L = 40$. It shows how the autocorrelation time increases as we approach the critical temperature from the ordered (cold) phase.

If $\tau_{\mathcal{O}} \ll 1$, then we obtain equation (13.39). If $\tau_{\mathcal{O}} \gg 1$

$$\begin{aligned}(\delta\mathcal{O})^2 &\approx \frac{2\tau_{\mathcal{O}}}{n-1}\left(\langle\mathcal{O}^2\rangle - \langle\mathcal{O}\rangle^2\right)\\ &\approx \frac{1}{(n/2\tau_{\mathcal{O}})}\left(\langle\mathcal{O}^2\rangle - \langle\mathcal{O}\rangle^2\right)\\ &\approx \frac{1}{n_{\mathcal{O}}-1}\left(\langle\mathcal{O}^2\rangle - \langle\mathcal{O}\rangle^2\right) \qquad (13.41)\end{aligned}$$

which is nothing but equation (13.39) for $n_{\mathcal{O}}$ independent measurements (we assumed that $1 \ll n_{\mathcal{O}} \ll n$). The above relation is consistent with our assumption that measurements become independent after time $\sim 2\tau_{\mathcal{O}}$.

In some cases, the straightforward application of equations (13.41) is not convenient. This happens when, measuring the autocorrelation time according to the discussion in section 13.5, becomes laborious and time consuming. Moreover, one has to compute the errors of observables that are functions of correlated quantities, like in the case of the magnetic susceptibility (13.30). The calculation requires the knowledge of quantities that are not defined on one spin configuration, like $\langle m\rangle$ and $\langle m^2\rangle$

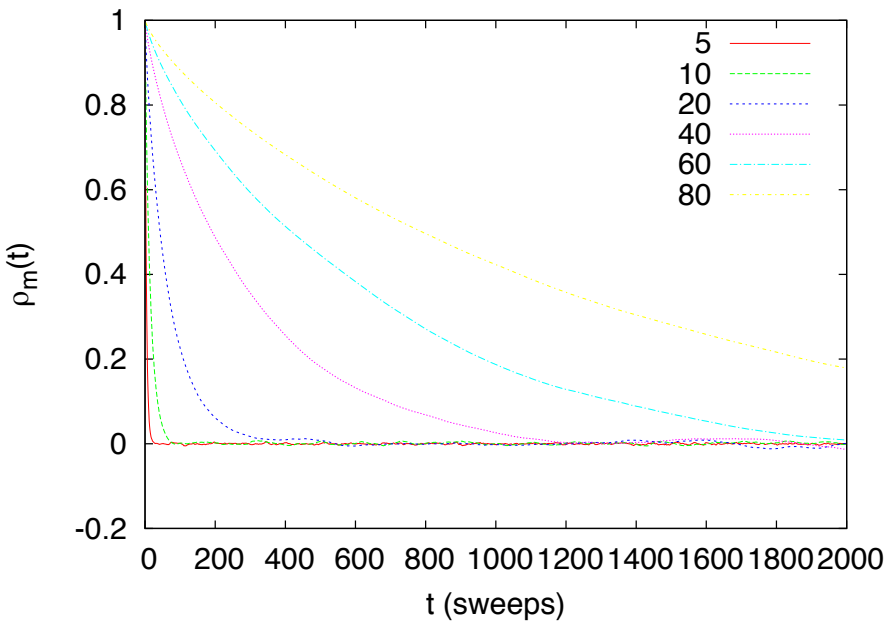



Figure 13.19: The autocorrelation function for the Ising model for $\beta = 0.4407 \approx \beta_c$ and different $L$. We observe the increase of the autocorrelation time with the system size in the critical region.

(or $(m_i - \langle m \rangle)$ on each configuration $i$). After these are calculated on the sample, the error $\delta\chi$ is not a simple function of $\delta\langle m \rangle$ and $\delta\langle m^2 \rangle$. This is because of the correlation between the two quantities and the well known formula of error propagation $(\delta(\langle m^2 \rangle - \langle m \rangle^2))^2 = (\delta\langle m^2 \rangle)^2 + (\delta\langle m \rangle^2)^2$ cannot be applied.

### 13.6.2 Jackknife

The simplest solution to the problems arising in the calculation of statistical errors discussed in the previous section is to divide a sample into blocks or *bins*. If one has $n$ measurements, she can put them in $n_b$ "bins" and each bin is to be taken as an independent measurement. This will be true if the number of measurements per bin $b = (n/n_b) \gg \tau_{\mathcal{O}}$. If $\mathcal{O}_i^b$ $i = 0, \ldots, n_b - 1$ is average value of $\mathcal{O}$ in the bin $i$, then the error is given by (13.39)

$$(\delta\mathcal{O})^2 = \frac{1}{n_b - 1}\left\{\frac{1}{n_b}\sum_{i=0}^{n_b-1}(\mathcal{O}_i^b - \langle \mathcal{O}^b \rangle)^2\right\} \tag{13.42}$$

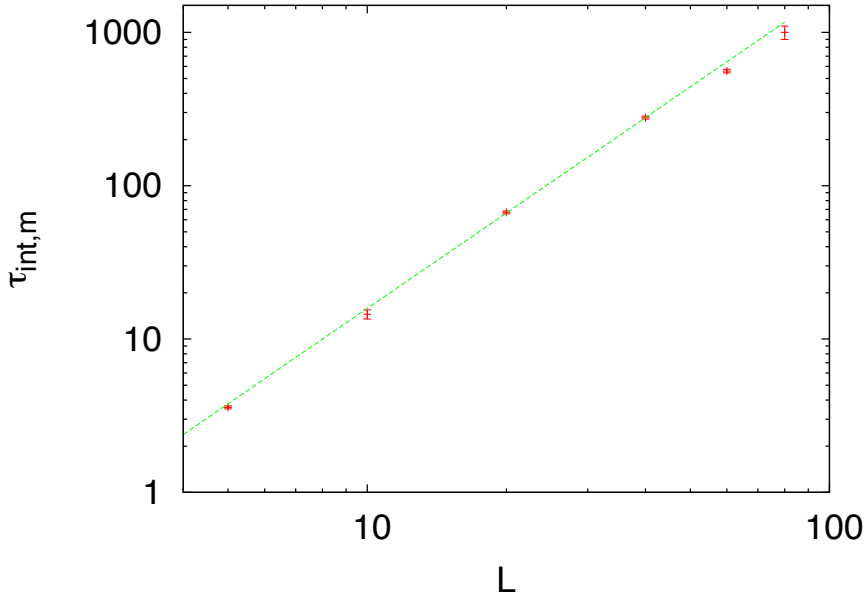



Figure 13.20: The integrated autocorrelation time $\tau_{\text{int},m}$ for $\beta = \beta_c$ in a logarithmic scale. The continuous line is the fit to $0.136(10)L^{2.067(21)}$. The expected result from the bibliography is $z = 2.1665(12)$ and the difference is a finite size effect.

This is the *binning* or *blocking* method and it is quite simple in its use. Note that quantities, like the magnetic susceptibilities, are calculated in each bin as if the bin were an independent sample. Then the error is easily calculated by equation (13.42). If the bin is too small and the samples are not independent, then the error is underestimated by a factor of $2\tau_{\mathcal{O}}/(n_b - 1)$ (see equation (13.40)). The bins are statistically independent if $b \sim 2\tau_{\mathcal{O}}$. If $\tau_{\mathcal{O}}$ is not a priori known we compute the error (13.42) by decreasing the number of bins $n_b$. When the error is not increasing anymore and takes on a constant value, then the calculation converges to the true statistical error.

But the method of choice in this book is the *jackknife method*. It is more stable and more reliable, especially if the sample is small. The basic idea is similar to the binning method. The difference is that the bins are constructed in a different way and equation (13.42) is slightly modified. The data is split in $n_b$ bins which contain $b = n - (n/n_b)$ elements as follows: The bin $j$ contains the part of the sample obtained after we we *erase* the contents of the $j$-th bin of the binning method from the full sample $\mathcal{O}_0, \ldots, \mathcal{O}_{n-1}$. The procedure is depicted in figure

13.21. We calculate the average value of $\mathcal{O}$ in each bin and we obtain

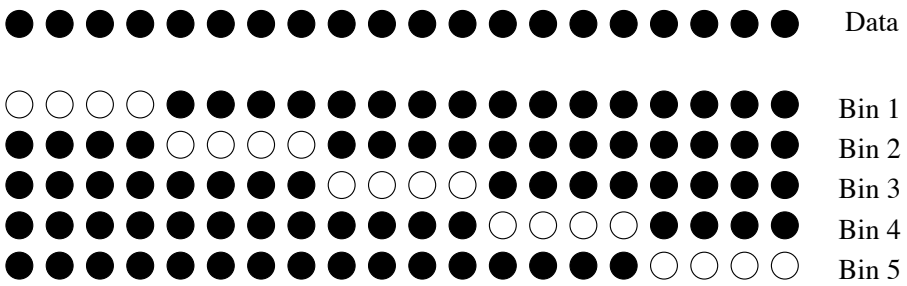



Figure 13.21: The jackknife method applied on a sample of $n = 20$ measurements. The data is split to $n_b = 5$ bins and each bin contains $b = n - (n/n_b) = 20 - 4 = 16$ measurements (the black disks). We calculate the average value $\mathcal{O}_i^b$ in each bin and, by using them, we calculate the error $\delta\mathcal{O} = \sqrt{n_b\left(\langle(\mathcal{O}^b)^2\rangle - \langle\mathcal{O}^b\rangle^2\right)}$.

$\mathcal{O}_0^b, \mathcal{O}_1^b, \ldots, \mathcal{O}_{n_b-1}^b$. Then the statistical error in the measurement of $\mathcal{O}$ is

$$(\delta\mathcal{O})^2 = \sum_{j=0}^{n_b-1} \left(\mathcal{O}_j^b - \langle\mathcal{O}^b\rangle\right)^2 = n_b\left(\langle(\mathcal{O}^b)^2\rangle - \langle\mathcal{O}^b\rangle^2\right) . \tag{13.43}$$

In order to determine the error, one has to vary the number of bins and check for the convergence of (13.43), like in the case of the binning method.

For more details and proofs of the above statements, the reader is referred to the book of Berg [5]. Appendix 13.8.1 provides examples and a program for the calculation of jackknife errors.

### 13.6.3 Bootstrap

Another useful method for the estimation of statistical errors is the *bootstrap method*. Suppose that we have $n$ *independent* measurements. From these we create $n_S$ random samples as follows: We choose one of the $n$ measurements with equal probability. We repeat $n$ times using the *same* set of $n$ measurements - i.e. by putting the chosen measurements back to the sample. This means that on the average $\sim 1 - 1/e \approx 63\%$ of the sample will consist of the same measurements. In each sample $i = 0, \ldots, n_S - 1$ we calculate the average values $\mathcal{O}_i^S$ and from those

$$\langle\mathcal{O}^S\rangle = \frac{1}{n_S}\sum_{i=0}^{n_S-1} \mathcal{O}_i^S , \tag{13.44}$$

and

$$\langle \left(\mathcal{O}^{S}\right)^{2} \rangle = \frac{1}{n_S} \sum_{i=0}^{n_S-1} \left(\mathcal{O}_i^{S}\right)^{2} . \tag{13.45}$$

The estimate for the error in $\langle \mathcal{O} \rangle$ is[37]

$$(\delta \mathcal{O})^2 = \langle \left(\mathcal{O}^{S}\right)^{2} \rangle - \langle \mathcal{O}^{S} \rangle^2 . \tag{13.46}$$

We stress that the above formula gives the error for independent measurements. If we have non negligible autocorrelation times, then we must use the correction

$$(\delta \mathcal{O})^2 = (1 + 2\tau_{\mathcal{O}}) \left( \langle \left(\mathcal{O}^{S}\right)^{2} \rangle - \langle \mathcal{O}^{S} \rangle^2 \right) \tag{13.47}$$

Appendix 13.8.2 discusses how to use the bootstrap method in order to calculate the true error $\delta \mathcal{O}$ without an a priori knowledge of $\tau_{\mathcal{O}}$. For more details, the reader is referred to the articles of Bradley Efron [59]. In appendix 13.8.2 you will find examples and a program that implements the bootstrap method.

## 13.7 Appendix: Autocorrelation Function

This appendix discusses the technical details of the calculation of the autocorrelation function (13.31) and the autocorrelation time given by equations (13.32) and (13.34). The programs can be found in the directory `Tools` in the accompanying software.

If we have a finite sample of $n$ measurements $\mathcal{O}(0), \mathcal{O}(1), \ldots, \mathcal{O}(n-1)$, then we can use the following estimator for the autocorrelation function, given by equation (13.31),

$$\rho_{\mathcal{O}}(t) = \frac{1}{\rho_0} \frac{1}{n-t} \sum_{t'=0}^{n-1-t} (\mathcal{O}(t') - \langle \mathcal{O} \rangle_0)(\mathcal{O}(t'+t) - \langle \mathcal{O} \rangle_t) , \tag{13.48}$$

where the average values are computed from the equations[38]

$$\langle \mathcal{O} \rangle_0 \equiv \frac{1}{n-t} \sum_{t'=0}^{n-1-t} \mathcal{O}(t') \qquad \langle \mathcal{O} \rangle_t \equiv \frac{1}{n-t} \sum_{t'=0}^{n-1-t} \mathcal{O}(t'+t) . \tag{13.49}$$

[37]Notice that the right hand side of equation (13.46) is *not* divided by $1/(n_s - 1)$.

[38]Alternatively one can take $\langle \mathcal{O} \rangle_0 = \langle \mathcal{O} \rangle_t = (1/n) \sum_{t'=0}^{n} \mathcal{O}(t')$ without noticeable difference for $t \ll n$. The choice in (13.48) results in a more accurate calculation and smaller finite sample effects. The choice (13.48), instead of $\rho_{\mathcal{O}}(t) \propto (1/(n-t)) \sum_{t'=0}^{n-1-t} \mathcal{O}(t')\mathcal{O}(t'+t) - \langle \mathcal{O} \rangle_0 \langle \mathcal{O} \rangle_t$, has smaller roundoff errors.

The constant $\rho_0$ is chosen so that $\rho_{\mathcal{O}}(0) = 1$.

The program for the calculation of (13.48) and the autocorrelation time (13.34) is listed below. It is in the file `autoc.f90` and you should read the comments embedded in the code for explanations of the most important steps.

```
!=====================================================
!file: autoc.f90
MODULE rho_function
 implicit none
 SAVE
 integer                    :: NMAX,tmax
 character(200)             :: prog
 CONTAINS
!-----------------------------------------------------
!rho is the unnormalized autocorrelation function at t:
  real(8) function rho(x,ndat,t)
   implicit none
   integer                  :: ndat,t
   real(8),dimension(0:)    :: x
   integer                  :: n,t0
   real(8)                  :: xav0,xavt,r
!-----------------------------------------
   n=ndat-t
   if(n<1) call locerr('rho: n<1')
!Calculate the two averages: xav0=<x>_0, xavt=<x>_t
   xav0 = SUM( x(0:n-1  )) / n
   xavt = SUM( x(t:n-1+t)) / n
   rho  = SUM((x(0:n-1)-xav0)*(x(t:n-1+t)-xavt))/n
  end function rho
!-----------------------------------------------------
  subroutine locerr(errmes)
   implicit none
   character(*) :: errmes
   write(0,'(A,A)'),TRIM(prog),':',TRIM(errmes),' Exiting....'
   stop 1
  end subroutine locerr
END MODULE rho_function
!=====================================================
program autocorrelations
 USE rho_function
 implicit none
 real(8),allocatable,dimension(:) :: r,tau,x
 real(8)                          :: norm
 integer                          :: i,ndat,t,tcut,chk
!-----------------------------------------------------
!Default values for max number of data and max time for
```

```
!rho and tau:
 NMAX=2000000;tmax=1000 !NMAX=2e6 requires ~ 2e6*8=16MB
 call get_the_options
 ALLOCATE(x(0:NMAX-1),STAT=chk)
 if(chk > 0) call locerr('Not enough memory for x')
 ndat=0
 do while ( ndat < NMAX)
  read(*,*,END=101)x(ndat)
  ndat = ndat+1
 enddo !
101 continue
 if(ndat >= NMAX) write(0,'(3A,I14,A,I14)')  &
      '# ',TRIM(prog),                       &
      ': Warning: read ndat=', ndat,         &
      ' and reached the limit: ',NMAX
!We decrease tmax if it is comparable or large of ndat
 if(tmax > (ndat/10) ) tmax = ndat/10
!r(t) stores the values of the autocorrelation function rho(t)
 ALLOCATE(r(0:tmax-1))
 do t=0,tmax-1
  r(t) = rho(x,ndat,t)
 enddo
 norm  = 1.0D0/r(0); r = norm*r
!tau(t) stores integrated autocorrelation times with tcht=t
 ALLOCATE(tau(0:tmax-1))
 do tcut=0,tmax-1
  tau(tcut)=0.0D0
  do t=0,tcut
   tau(tcut) = tau(tcut)+r(t)
  enddo
 enddo
!Output:
 print '(A)','# ==========================================='
 print '(A)','# Autoc function rho and int autoc time tau  '
 print '(A,I12,A,I8)','# ndat= ',ndat,'  tmax= ',tmax
 print '(A)','# t          rho(t)              tau(tcut=t)  '
 print '(A)','# ==========================================='
 do t=0,tmax-1
  print '(I8,2G28.17)',t,r(t),tau(t)
 enddo
end program autocorrelations
!=====================================================
subroutine get_the_options
 use rho_function
 use getopt_m    !from getopt.f90
 implicit none
 call getarg(0,prog)

 do
```

```
  select case( getopt( "-ht:n:" ))
  case( 't' )
   read(optarg,*)tmax
  case( 'n' )
   read(optarg,*)NMAX
  case( 'h' )
   call usage
  case( '?' )
   print *, 'unknown option    ', optopt
   stop
  case( char(0)) ! done with options
   exit
  case( '-' )     ! use -- to exit from options
   exit
  case default
   print *, 'unhandled option ', optopt
  end select
 enddo

end subroutine get_the_options
!=====================================================
subroutine usage
 use rho_function
 implicit none
 print '(3A)','Usage: ',TRIM(prog),&
              ' [-t <maxtime>] [-n <ndat>]'
 print '( A)','      Reads data from stdin (one column) and'
 print '( A)','       computes autocorrelation function and'
 print '( A)','       integrated autocorrelation time.'
 stop
end subroutine usage
!=====================================================
```

The calculation of the autocorrelation function is put in a separate module `rho_function` which can be used by any of your programs. After the statement `CONTAINS` we can add code for functions and subroutines which can be accessed[39] by any program unit that uses the module. The module makes global variables, like `NMAX`, `tmax` and `prog`, accessible to all program units that use the module.

The compilation is done with the commands

```
> gfortran -O2 getopt.f90 autoc.f90 -o autoc
```

If our data is written in a file named `data` in one column, then the

[39]The `explicit interface` of the function or the subroutine is known to all program units that use the module.

calculation of the autocorrelation function and the autocorrelation time is done with the command

```
> cat data | ./autoc > data.rho
```

The results are written to the file `data.rho` in three columns. The first one is the time $t$, the second one is $\rho_{\mathcal{O}}(t)$ and the third one is $\tau_{\text{int},\mathcal{O}}(t)$ (equation (13.35)). The corresponding plots are constructed by the `gnuplot` commands:

```
gnuplot> plot "data.rho" using 1:2 with lines
gnuplot> plot "data.rho" using 1:3 with lines
```

If we wish to increase the maximum number of data `NMAX` or the maximum time `tmax`, then we use the options `-n` and `-t` respectively:

```
> cat data | autoc -n 20000000 -t 20000 > data.rho
```

For doing all the work at once using `gnuplot`, we can give the command:

```
gnuplot> plot "<./is -L 20 -b 0.4407 -s 1 -S 345 -n 400000|\
               grep -v '#'|awk '{print ($2>0)?$2:-$2;}'  |\
               autoc -t 500" using 1:2 with lines
```

The above command is long and it is broken into 3 lines for better printing. You can type it in one line by removing the trailing \.

A script that works out many calculations together is listed below. It is in the file `autoc_L` and computes the data shown in figure 13.19.

```
#!/bin/tcsh -f

set nmeas  = 2100000
set Ls     = (5 10 20 40 60 80)
set beta   = 0.4407
set tmax   = 2000
foreach L ($Ls)
 set N     = `awk -v L=$L 'BEGIN{print L*L}'`
 set rand  = `perl -e 'srand();print int(3000000*rand())+1;'`
 set out   = outL${L}b${beta}
 echo "Running L${L}b${beta}"
 ./is -L $L -b $beta -s 1 -S $rand -n $nmeas > $out
 echo "Autocorrelations L${L}b${beta}"
 grep -v '#' $out | \
```

```
  awk -v N=$N 'NR>100000{print ($2>0)?($2/N):(-$2/N)}'|\
  autoc -t $tmax  > $out.rhom
end
```

Then we give the gnuplot commands:

```
gnuplot>   plot  "outL5b0.4407.rhom" u 1:2 w lines title "5"
gnuplot> replot "outL10b0.4407.rhom" u 1:2 w lines title "10"
gnuplot> replot "outL20b0.4407.rhom" u 1:2 w lines title "20"
gnuplot> replot "outL40b0.4407.rhom" u 1:2 w lines title "40"
gnuplot> replot "outL60b0.4407.rhom" u 1:2 w lines title "60"
gnuplot> replot "outL80b0.4407.rhom" u 1:2 w lines title "80"
```

The plots in figure 13.17 are constructed in a similar way.

For the calculation of $\tau_m$ we do the following:

```
gnuplot> f(x) = c * exp(-x/t)
gnuplot> set log y
gnuplot> plot [:1000] "outL40b0.4407.rhom" u 1:2 with lines
gnuplot> c = 1 ; t = 300
gnuplot> fit [150:650] f(x) "outL40b0.4407.rhom" u 1:2 via c,t
gnuplot> plot [:1000] "outL40b0.4407.rhom" u 1:2 w lines,f(x)
gnuplot> plot [:]     "outL40b0.4407.rhom" u 1:3 w lines
```

where in the last line we compute $\tau_{\text{int},m}$. The `fit` command is just an example and one should try different fitting ranges. The first plot command shows graphically the approximate range of the exponential falloff of the autocorrelation function. We should vary the upper and lower limits of the fitting range until the value of $\tau_m$ stabilizes and the[40] $\chi^2/\text{dof}$ is minimized[41]. The $\chi^2/\text{dof}$ of the fit can be read off from the output of the command `fit`

```
.....
degrees of freedom    (FIT_NDF)                       : 449
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf) : ↩
   0.000939201
```

[40]For the data $\{(x_i, y_i)\}$, $i = 1, \ldots, n$ with error $\delta y_i$ which are fitted to $f(x; c, t) = c\,\mathrm{e}^{-x/t}$, the $\chi^2(c,t) = \sum_{i=1}^{n}(y_i - f(x_i; c, t))^2/\delta y_i^2$. The $\chi^2/\text{dof}$ is normalized to the number of degrees of freedom (dof = degrees of freedom= $n-2$) which is the number of data points $n$ used in the fit minus the number of fitting parameters (here $c$, $t$ which makes 2 parameters).

[41]The acceptable $\chi^2/\text{dof} \sim 1$. Since we don't calculate the errors of the autocorrelation functions, the $\chi^2/\text{dof}$ is not properly normalized. The program sets $\delta y_i = 1\ \forall i$.

```
variance of residuals(reduced chisquare) = WSSR/ndf: 8.82099e↩
   −07

Final set of parameters            Asymptotic Standard Error
=======================            =========================

c               = 0.925371         +/− 0.0003773     (0.04078%)
t               = 285.736          +/− 0.1141        (0.03995%)
.....
```

from the line "`variance of residuals`". From the next lines we read the values of the fitted parameters with their errors[42] and we conclude that $\tau_m = 285.7 \pm 0.1$. We stress that this is the *statistical error* of the fit for the given fitting range. But usually the largest contributions to the error come from systematic errors, which, in our case, are seen by varying the fitting range[43]. By trying different fitting ranges and using the criterion that the minimum $\chi^2/\text{dof}$ doubles its minimum value, we find that $\tau_m = 285(2)$.

In our case the largest systematic error comes from neglecting the effect of smaller autocorrelation times. These make non negligible contributions for small $t$.

By fitting to

$$f(t) = c\,\mathrm{e}^{-t/\tau}\,, \tag{13.50}$$

we have taken into account only the largest autocorrelation time.

One should take into account also the smaller autocorrelation times. In this case we expect that $\rho_m(t) \sim a_1\mathrm{e}^{-t/\tau_1} + a_2\mathrm{e}^{-t/\tau_2} + \ldots$. We find that the data for the autocorrelation function fit perfectly to the function

$$h(x) = a_1\mathrm{e}^{-t/\tau_1} + a_2\mathrm{e}^{-t/\tau_2} + a_3\mathrm{e}^{-t/\tau_3}\,. \tag{13.51}$$

As we can see in figures 13.22 and 13.23, the small $t$ fit is excellent and the result for the dominant autocorrelation time is $\tau_m \equiv \tau_1 = 286.3(3)$. The secondary autocorrelation times are $\tau_2 = 57(3)$, $\tau_3 = 10.5(8)$ which are considerably smaller that $\tau_1$.

The commands for the analysis are listed below:

[42]In the parentheses we see the confidence level. This is defined as the probability that the parameters are within the range defined by the error. For a correct calculation of the confidence level, every point should be weighted by its error - in our example this is not happening and this is the reason why the confidence level is so low. A value below 5% is too low and it indicates that the model needs corrections. It is also assumed that the distribution of the measurements is Gaussian and if not, the computed numerical values are only indicative.

[43]For a careful calculation, one should also try more functions that include corrections to the asymptotic behavior.

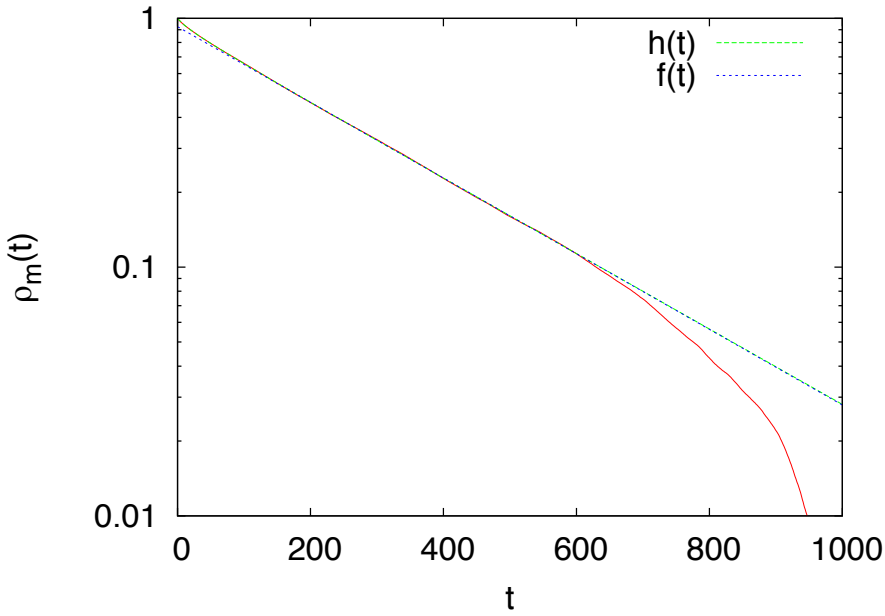



Figure 13.22: Fit of the autocorrelation function $\rho_m(t)$ to the functions $f(t) = c\,\mathrm{e}^{-t/\tau}$ and $h(t) = a_1\mathrm{e}^{-t/\tau_1} + a_2\mathrm{e}^{-t/\tau_2} + a_3\mathrm{e}^{-t/\tau_3}$. For large times $f(t) \approx h(t)$, but $h(t)$ is necessary in order to capture the small $t$ behavior. This choice results in $\tau_m = \tau = \tau_1$. The values of the parameters are given in the text. The vertical axes are in logarithmic scale.

```
gnuplot> h(x) = a1*exp(-x/t1) + a2*exp(-x/t2) + a3*exp(-x/t3)
gnuplot> a1 = 1;     t1 = 285; a2 = 0.04; t2 = 56; \
         a3 = 0.03; t3 = 10
gnuplot> fit [1:600] h(x) "outL40b0.4407.rhom" \
         using 1:2 via a1,t1,a2,t2,a3,t3
...
Final set of parameters            Asymptotic Standard Error
=======================            =========================
a1              = 0.922111         +/- 0.001046     (0.1135%)
t1              = 286.325          +/- 0.2354       (0.08221%)
a2              = 0.0462523        +/- 0.001219     (2.635%)
t2              = 56.6783          +/- 2.824        (4.982%)
a3              = 0.0300761        +/- 0.001558     (5.18%)
t3              = 10.5227          +/- 0.8382       (7.965%)
gnuplot> plot [:150][0.5:]   "outL40b0.4407.rhom" using 1:2 \
         with lines notit,h(x) ,f(x)
gnuplot> plot [:1000][0.01:] "outL40b0.4407.rhom" using 1:2 \
         with lines notit,h(x) ,f(x)
```

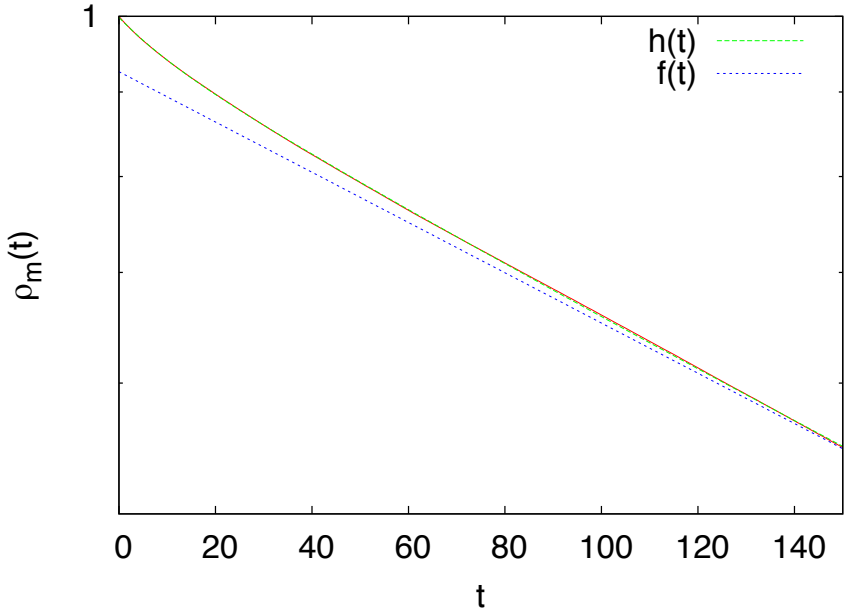



Figure 13.23: The plot of figure 13.22 for small times where the effect of smaller autocorrelation times is most clearly seen.

## 13.8 Appendix: Error Analysis

### 13.8.1 The Jackknife Method

In this section we present a program that calculates the errors using the jackknife method discussed in section 13.6.2. Figure 13.21 shows the division of the data into bins. For each bin we calculate the average value of the quantity $\mathcal{O}$ and then we use equation (13.43) in order to calculate the error. The program is in the file `jack.f90` which you can find in the directory `Tools` in the accompanying software. The program calculates $\langle \mathcal{O} \rangle$, $\delta\mathcal{O}$, $\chi \equiv \langle (\mathcal{O} - \langle \mathcal{O} \rangle)^2 \rangle$ and $\delta\chi$.

```
!=====================================================
!file: jack.f90
MODULE jack_function
 implicit none
 SAVE
 integer                    :: JACK,MAXDAT
 character(200)             :: prog
 CONTAINS
!-----------------------------------------------------
```

```
!jackknife function:
  subroutine jackknife(ndat,jack,x,&
       av0,er0,avchi,erchi)
   integer                 :: ndat,jack !local jack...
   real(8),dimension(0:)   :: x
   real(8)                 :: av0,er0,avchi,erchi
   integer                 :: i,j,binw,bin
   real(8),allocatable     :: O(:),chi(:)
!-----------------------------------------
   ALLOCATE(O(0:jack-1));ALLOCATE(chi(0:jack-1))
   O=0.0D0;chi=0.0D0;
   binw=ndat/jack
   if(binw<1)call locerr('jackknife: binw < 1')
!Average value:
   do i=0,ndat-1
    do j=0,jack-1
     if((i/binw) /= j) &
          O  (j) = O  (j) + x(i)
    enddo
   enddo
   O   = O  /(ndat-binw) !normalize
!Susceptibility:
   do i=0,ndat-1
    do j=0,jack-1
     if((i/binw) /= j) &
          chi(j) = chi(j) + (x(i)-O(j))*(x(i)-O(j))
    enddo
   enddo
   chi   = chi/(ndat-binw) !normalize
!---------------
   av0   = SUM(O)/jack;avchi=SUM(chi)/jack
   er0   = sqrt(SUM((O  -av0  )*(O  -av0  )))
   erchi = sqrt(SUM((chi-avchi)*(chi-avchi)))
!---------------
   DEALLOCATE(O);DEALLOCATE(chi)
  end subroutine jackknife
!-------------------------------------------------------
  subroutine locerr(errmes)
   implicit none
   character(*) :: errmes
   write(0,'(A,A)'),TRIM(prog),':',TRIM(errmes),' Exiting....'
   stop 1
  end subroutine locerr
END MODULE jack_function
!=======================================================
program jackknife_errors
 use jack_function
 implicit none
 integer              :: ndat,chk
```

```
 real(8)              :: O,dO,chi,dchi
 real(8),allocatable  :: x(:)
 MAXDAT=1000000;JACK=10
 call get_the_options
 ALLOCATE(x(0:MAXDAT-1),STAT=chk)
 if(chk > 0) call locerr('Not enough memory for x')
 ndat=0
 do while ( ndat < MAXDAT)
  read(*,*,END=101)x(ndat)
  ndat = ndat+1
 enddo
101 continue
 if(ndat >= MAXDAT) write(0,'(3A,I14,A,I14)') &
      '# ',TRIM(prog),                        &
      ': Warning: read ndat=', ndat,          &
      ' and reached the limit: ',MAXDAT
 call jackknife(ndat,JACK,x,O,dO,chi,dchi)
 print '(A,I14,A,I12,A)','#  NDAT = ',ndat,   &
       ' data. JACK = ',JACK,' groups'
 print '(A)','# <o>, chi= (<o^2>-<o>^2)'
 print '(A)','# <o> +/- err                 chi +/- err'
 print '(4G28.17)',O,dO,chi,dchi
end program jackknife_errors
!=====================================================
subroutine get_the_options
 use jack_function
 use getopt_m    !from getopt.f90
 implicit none
 call getarg(0,prog)

 do
  select case( getopt( "-hj:d:" ))
  case( 'j' )
   read(optarg,*)JACK
  case( 'd' )
   read(optarg,*)MAXDAT
  case( 'h' )
   call usage
  case( '?' )
   print *, 'unknown option   ', optopt
   stop
  case( char(0)) ! done with options
   exit
  case( '-' )    ! use -- to exit from options
   exit
  case default
   print *, 'unhandled option ', optopt
  end select
 enddo
```

```
end subroutine get_the_options
!==========================================
subroutine usage
 use jack_function
 implicit none
 print '(3A)','Usage: ',TRIM(prog),'  [options]'
 print '( A)','        -j  : No. jack groups Def. 10'
 print '( A)','        -d  : Max. no. of data points read'
 print '( A)','Computes <o>, chi= (<o^2>-<o>^2)'
 print '( A)','Data is in one column from stdin.'
 stop
end subroutine usage
```

For the compilation we use the command

```
> gfortran -O2 getopt.f90 jack.f90 -o jack
```

If we assume that our data is in one column in the file `data`, the command that calculates the jackknife errors using 50 bins is:

```
> cat data | jack -j 50
```

The program has set a maximum of `MAXDAT`=1,000,000 measurements. If we need to analyze more data, we have to use the switch `-d`. For example, for 2,000,000 measurements, use `-d 2000000`. The program reads data from the `stdin` and we can construct filters in order to do complicated analysis tasks. For example, the analysis of the magnetization produced by the output of the Ising model program can be done with the command:

```
> ./is -L 20 -b 0.4407 -s 1 -S 342 -n 2000000 | grep -v #  | \
  awk  -v L=20 '{print ($2>0)?($2/(L*L)):(-$2/(L*L))}'     | \
  ./jack -j 50 -d 2000000                          | grep -v #  | \
  awk  -v b=0.4407 -v L=20 '{print $1,$2,b*L*L*$3,b*L*L*$4}'
```

The command shown above can be written in one line by removing the backslashes ('\') at the end of each line. Let's explain it in detail: The first line runs the program `is` for the Ising model with $N = L \times L = 20 \times 20$ lattice sites (`-L 20`) and $\beta = 0.4407$ (-b 0.4407). It starts the simulation from a hot configuration (`-s 1`) and makes 2,000,000 measurements (`-n 2000000`). The command `grep -v` filters out the comments from the output of the program, which are lines starting with a #. The second line calls `awk` and defines the `awk` variable `L` to be equal to 20 (`-v L=20`). For

each line in its input, it prints the absolute value of the second column ($2) divided by the number of lattice sites L*L. The third line makes the jackknife calculation of the average values of $\langle m \rangle$ and $\langle (m - \langle m \rangle)^2 \rangle$ with their errors using the program jack. The comments of the output of the command jack are removed with the command grep -v. The fourth line is needed only for the calculation of the magnetic susceptibility, using equation (13.30). There, we need to multiply the fluctuations $\langle (m - \langle m \rangle)^2 \rangle$ and their error by the factor $\beta N = \beta L^2$ in order to obtain $\chi$.

### 13.8.2 The Bootstrap Method

In this subsection we present a program for the calculation of the errors using the bootstrap method according to the discussion in section 13.6.3. The program is in the file boot.f90:

```
!=====================================================
!file: boot.f90
MODULE boot_function
 implicit none
 SAVE
 integer                    :: SAMPLES,MAXDAT
 character(200)             :: prog
 integer                    :: seed
 CONTAINS
!-----------------------------------------------------
!jackknife function:
  subroutine bootstrap(ndat,samples,x,&
       av0,er0,avchi,erchi)
   integer                  :: ndat,samples !local samples...
   real(8),dimension(0:)    :: x
   real(8)                  :: av0,er0,avchi,erchi
   integer                  :: i,j,k
   real(8),allocatable      :: O(:),O2(:),chi(:)
!-----------------------------------------
   ALLOCATE(O(0:samples-1));ALLOCATE(O2(0:samples-1));
   ALLOCATE(chi(0:samples-1))
   O=0.0D0;O2=0.0D0;chi=0.0D0;
   do  j=0,samples-1
    do i=0,ndat   -1
     k  = INT(ndat*drandom()) !0,...,ndat-1
     O (j) = O (j) + x(k)
     O2(j) = O2(j) + x(k)*x(k)
    enddo
    O  (j) = O(j)/ndat; O2(j) = O2(j)/ndat
    chi(j) = O2(j)-O(j)*O(j)
```

```
   enddo
!-----------------
   av0   = SUM(0)/samples;avchi=SUM(chi)/samples
   er0   = sqrt(SUM((0  -av0  )*(0  -av0  ))/samples)
   erchi = sqrt(SUM((chi-avchi)*(chi-avchi))/samples)
!compute the real av0:
   av0   = SUM(x(0:ndat-1))/ndat
!-----------------
   DEALLOCATE(0);DEALLOCATE(chi)
  end subroutine bootstrap
!-------------------------------------------------------
  real(8) function drandom()
   implicit none
   integer,parameter :: a = 16807
   integer,parameter :: m = 2147483647
   integer,parameter :: q = 127773
   integer,parameter :: r = 2836
   real(8),parameter :: f = (1.0D0/m)
   integer           :: p
   real(8)           :: dr
101 continue
   p       = seed/q
   seed    = a*(seed- q*p) - r*p
   if(seed .lt. 0) seed = seed + m
   dr      = f*seed
   if( dr .le. 0.0D0 .or. dr .ge. 1.0D0) goto 101
   drandom = dr
  end function drandom
!-------------------------------------------------------
  subroutine locerr(errmes)
   implicit none
   character(*) :: errmes
   write(0,'(A,A)'),TRIM(prog),':',TRIM(errmes),' Exiting....'
   stop 1
  end subroutine locerr
END MODULE boot_function
!=======================================================
program bootstrap_errors
 use boot_function
 implicit none
 integer              :: ndat,chk
 real(8)              :: 0,d0,chi,dchi
 real(8),allocatable  :: x(:)
 MAXDAT=1000000;SAMPLES=1000
 call get_the_options
 ALLOCATE(x(0:MAXDAT-1),STAT=chk)
 if(chk > 0) call locerr('Not enough memory for x')
 ndat=0
 do while ( ndat < MAXDAT)
```

```
  read(*,*,END=101)x(ndat)
  ndat = ndat+1
 enddo
101 continue
 if(ndat >= MAXDAT) write(0,'(3A,I14,A,I14)') &
      '# ',TRIM(prog),                         &
      ': Warning: read ndat=', ndat,           &
      ' and reached the limit: ',MAXDAT
 open (28, file="/dev/urandom", access="stream",&
  form="unformatted")
 read (28) seed
 seed = ABS(seed)
 close(28)
 call bootstrap(ndat,SAMPLES,x,O,dO,chi,dchi)
 print '(A,I14,A,I12,A)',&
  '#  NDAT = ',ndat,' data. SAMPLES = ',SAMPLES,' groups'
 print '(A              )',&
  '# <o>, chi= (<o^2>-<o>^2)'
 print '(A              )',&
  '# <o> +/- err                              chi +/- err'
 print '(4G28.17)',O,dO,chi,dchi
end program bootstrap_errors
!=====================================================
subroutine get_the_options
 use boot_function
 use getopt_m    !from getopt.f90
 implicit none
 call getarg(0,prog)

 do
  select case( getopt( "-hs:d:" ))
  case( 's' )
   read(optarg,*)SAMPLES
  case( 'd' )
   read(optarg,*)MAXDAT
  case( 'h' )
   call usage
  case( '?' )
   print *, 'unknown option    ', optopt
   stop
  case( char(0)) ! done with options
   exit
  case( '-' )    ! use -- to exit from options
   exit
  case default
   print *, 'unhandled option ', optopt
  end select
 enddo
```

```
end subroutine get_the_options
!==========================================
subroutine usage
 ...
end subroutine usage
!==========================================
```

For the compilation we use the command

```
> gfortran -O2 getopt.f90 boot.f90 -o boot
```

If our data is in one column in the file `data`, then the command that calculates the errors using 500 samples is:

```
> cat data | boot -s 500
```

The maximum number of measurements is set to 1,000,000 as in the `jack` program. For more measurements we should use the `-d` switch, e.g. for 2,000,000 measurements use `-d 2000000`. For the analysis of the magnetization from the output of the program `is` we can use the following command:

```
>  is -L 20 -b 0.4407 -s 1 -S 342 -n 2000000 | grep -v #  | \
  awk  -v L=20 '{print ($2>0)?($2/(L*L)):(-$2/(L*L))}'   | \
  boot -s 1000  -d 2000000                  | grep -v #  | \
  awk  -v b=0.4407 -v L=20 '{print $1,$2,b*L*L*$3,b*L*L*$4}'
```

### 13.8.3 Comparing the Methods

In this subsection we will compute errors using equation (13.40), the jackknife method (13.43) and the bootstrap method (13.47). In order to appreciate the differences, we will use data with large autocorrelation times. We use the Metropolis algorithm on the Ising model with $L = 40$, $\beta = 0.4407 \approx \beta_c$ and measure the magnetization per site (13.28). We take $1,000,000$ measurements using the commands:

```
> ./is -L 40 -b 0.4407 -s 1 -S 5434365 -n 1000000 \
                              > outL40b0.4407.dat &
> grep -v # outL40b0.4407.dat | \
  awk -v L=40 '{if($2<0){$2=-$2};print $2/(L*L)}' \
                              > outL40b0.4407.m
> cat outL40b0.4407.m | autoc -t 10000 -n 1000000 \
```

```
> outL40b0.4407.rhom
```

The file `outL40b0.4407.m` has the measurements of the magnetization in one column and the file `outL40b0.4407.rhom` has the autocorrelation function and the integrated autocorrelation time as described after page 548. We obtain $\tau_m = 286.3(3)$. The integrated autocorrelation time is found to be $\tau_{\text{int,m}} = 254(1)$.

The expectation value is $\langle m \rangle = 0.638682$. The application of equation (13.39), valid for independent measurements, gives the (underestimated) error $\delta_c m = 0.00017$. Using equation (13.40) we obtain $\delta m = \sqrt{1+2\tau}\delta_c m \approx 0.004$. The error of the magnetic susceptibility cannot be calculated this way.

$$\langle m \rangle = 0.639 \pm 0.004 \equiv 0.639(4) \tag{13.52}$$

For the calculation of the error of the magnetic susceptibility we have

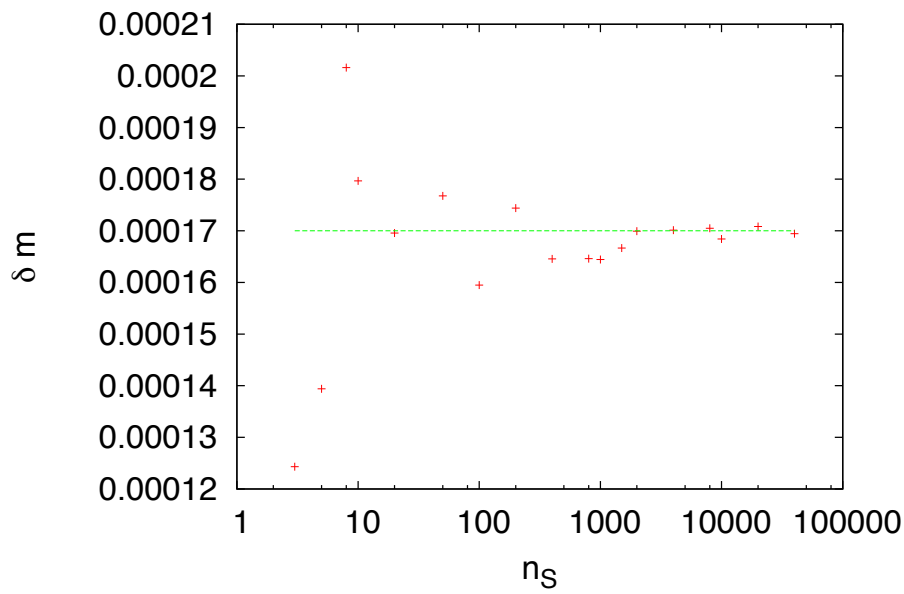



Figure 13.24: The error $\delta m$ calculated using the bootstrap method as a function of the number of samples $n_S$. We observe a very fast convergence to the value obtained by equation (13.39) $\delta_c m = 0.00017$.

to resort to the jackknife or to the bootstrap method. The latter is applied initially using a variable number of samples $n_S$ so that the optimal number of samples is be determined. Figure 13.24 shows the results for

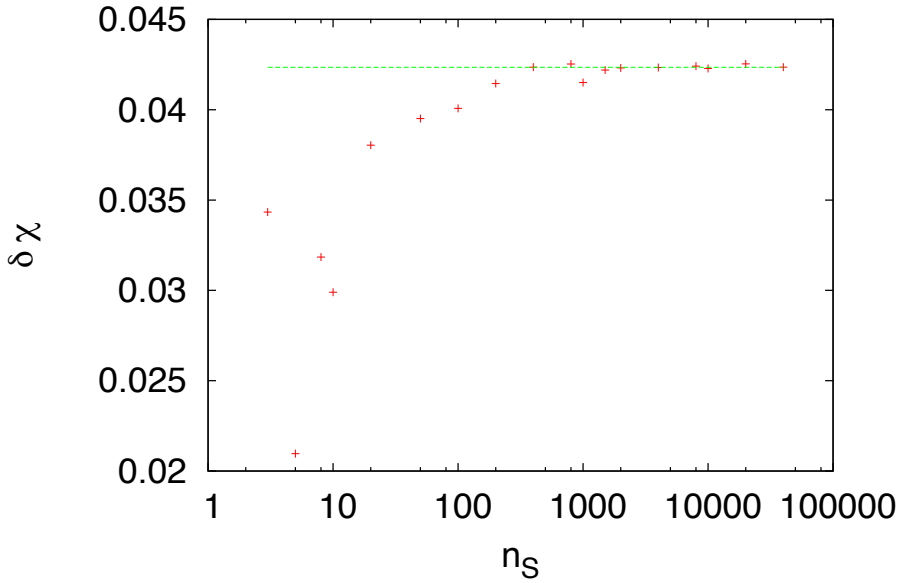



Figure 13.25: The error $\delta\chi$ of the magnetic susceptibility calculated using the bootstrap method as a function of the number of samples $n_S$. We observe convergence for $n_S > 1000$ to the value $\delta_c\chi = 0.0435$.

the magnetization. We observe a very fast convergence to $\delta_c m = 0.00017$ for quite small number of samples. The analysis could have safely used $n_S = 100$. In the case of the magnetic susceptibility, convergence is slower, but we can still use $n_S = 500$. We obtain $\chi = 20.39$ and $\delta_c\chi = 0.0435$. The error assumes independent measurements, something that is not true in our case. We should use the correction factor $\sqrt{1 + 2\tau_m}$ which gives $\delta\chi = 1$. Therefore

$$\chi = 20 \pm 1 \equiv 20(1) \tag{13.53}$$

We note that the error is quite large, which is because we have few independent measurements: $n/(2\tau_m) \approx 1,000,000/(2 \times 286) \approx 1750$. The a priori knowledge of $\tau_m$ is necessary in this calculation. In the case of the jackknife method, the calculation can proceed without an priori knowledge of $\tau_m$. The errors are calculated for a variable number of bins $n_b$. Figure 13.26 shows the results for the magnetization. When $n_b = n$ the samples consist of all the measurements except one. Then the error is equal to the error calculated using the standard deviation formula and it is underestimated by the factor $\sqrt{1 + 2\tau_m}$. This is shown in figure 13.26, where we observe a slow convergence to the value $\delta_c m = 0.00017$. The

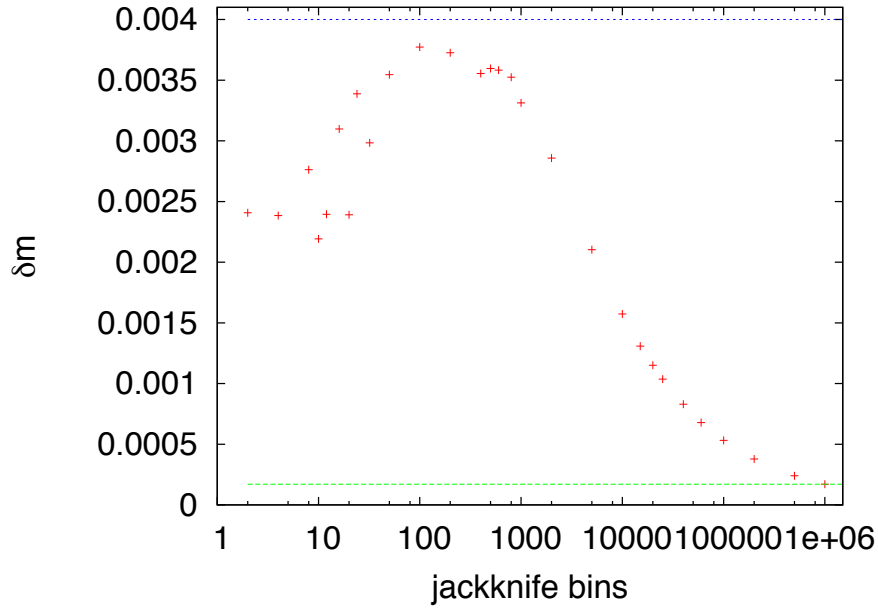



Figure 13.26: The error $\delta m$ calculated using the jackknife method as a function of the number of bins $n_b$. Convergence is observed for $100 < n_b < 800$ to $\delta m = 0.0036$. The plot shows that as we approach the limit $n_b = n$, the error approaches the value calculated by equation (13.39) $\delta_c m = 0.00017$. The horizontal lines correspond to the values $\delta_c m$ and $\sqrt{1+2\tau_m}\delta_c m \approx 0.004$ where $\tau_m = 286.3$. The ratio $\delta m/\delta_c m \approx \sqrt{1+2\tau_m}$.

effect of the autocorrelations vanishes when we delete (bin width) $\approx 2\tau_m$ measurements from each bin. This happens when $n_b \approx n/(\text{bin width}) = n/(2\tau_m) = 1,000,000/572 \approx 1750$. Of course this an order of magnitude estimate and a careful study is necessary in order to determine the correct value for $n_b$. Figure 13.26 shows that the error converges for $100 < n_b < 800$ to the value $\delta m = 0.0036$, which is quite close to the value $\sqrt{1+2\tau_m}\delta_c m \approx 0.004$. We note that, by using a small number $n_b \approx 20-40$, we obtain an acceptable estimate, a rule of the thumb that can be used for quick calculations.

Similar results are obtained for the magnetic susceptibility $\chi$, where the error converges to the value $\delta\chi = 0.86$, in accordance with the previous estimates. For $n_b \to n$ the error converges to the underestimated error $\delta_c\chi = 0.0421$. We can use the bootstrap method, in a similar way to the jackknife method, in order to determine the real error

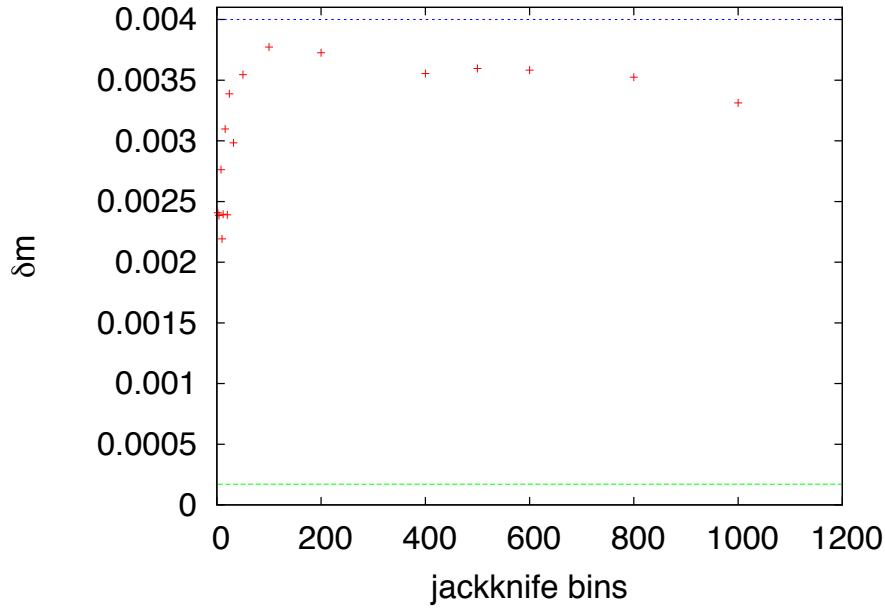



Figure 13.27: Figure 13.26 magnified in the region of the plateau in the values of $\delta m$. The horizontal lines correspond to the values $\delta_c m$ and $\sqrt{1+2\tau_m}\delta_c m \approx 0.004$ where $\tau_m = 286.3$. The ratio $\delta m/\delta_c m \approx \sqrt{1+2\tau_m}$.

$\delta m$, $\delta\chi$ without calculating $\tau_m$ directly. The data is split into $n_b$ bins[44], whose bin width is (bin width) $= n/n_b$. Each bin gives an independent measurement, the average value of the observable in this bin. We take $n_S = 1000$ and we apply the bootstrap method on the sample of the $n_b$ measurements. Figures 13.30 and 13.32 show the results as a function of the (bin width). When (bin width) $\approx 2\tau_m$ the measurements become practically independent and we should obtain the correct error. Figure 13.30 shows the results for the magnetization where for (bin width)=1 we obtain $\delta_c m = 0.00017$. When the (bin width)$> 2\tau_m$ the error becomes $\delta m = 0.0036$. This is read off from the plateau for $1100 <$ (bin width) $< 16000$ and the result is consistent with the one obtained by the jackknife method. Figure 13.30 shows the results for the magnetic susceptibility. Again, for (bin width)=1 we obtain $\delta_c\chi = 0.0421$. When the (bin width)$> 2\tau_m$ the error becomes $\delta\chi = 0.615$. This is read off from the plateau for $500 <$ (bin width) $< 1000$ and the result is consistent

[44]Note that if $n_b$ is too small, then we have a systematic error in the value of $\chi$. The results, as well as the error, must be monitored so that the correct range in the values of $n_b$ is found in which the error is independent of $n_b$.

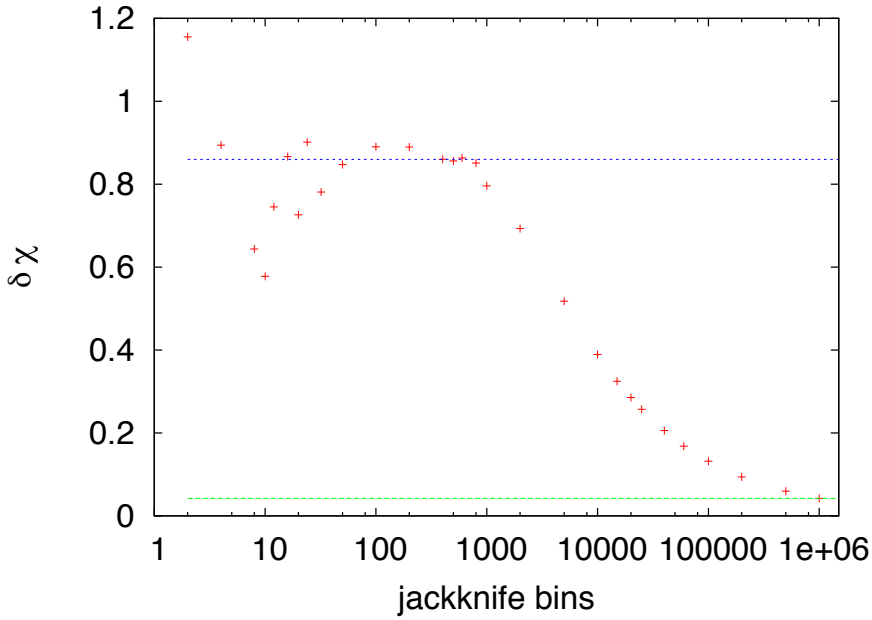



Figure 13.28: The error $\delta\chi$ calculated using the jackknife method as a function of the number of bins $n_b$. Convergence is observed for $100 < n_b < 800$ to $\delta\chi = 0.86$. The plot shows that as we approach the limit $n_b = n$, the error approaches the same value $\delta_c\chi = 0.0421$ that would have been obtained if we had falsely considered the measurements to be independent. These values are very close to the ones obtained using the bootstrap method. The values $\delta\chi$ and $\delta_c\chi$ are shown in the plots by the two horizontal lines. The ratio $\delta\chi/\delta_c\chi \approx \sqrt{1+2\tau_m}$.

with the one obtained by the jackknife method.

The above calculations can be reversed and used for the calculation of the autocorrelation time. By computing the underestimated error $\delta_c\mathcal{O}$ and the true error $\delta\mathcal{O}$ using one of the methods described above, we can calculate $\tau_m$ using the relation $\delta\mathcal{O}/\delta_c\mathcal{O} = \sqrt{1+2\tau_{\mathcal{O}}}$. Therefore

$$\tau_m = \frac{1}{2}\left(\left(\frac{\delta m}{\delta_c m}\right)^2 - 1\right) = \frac{1}{2}\left(\left(\frac{\delta\chi}{\delta_c\chi}\right)^2 - 1\right) = \ldots \tag{13.54}$$

By calculating $\tau_m$ using all the methods described here, these relations can also be used in order to check the analysis for self-consistency and see if they agree. This is not always a trivial work since a system may have many autocorrelation times which influence each observable in a different way (fast modes, slow modes).

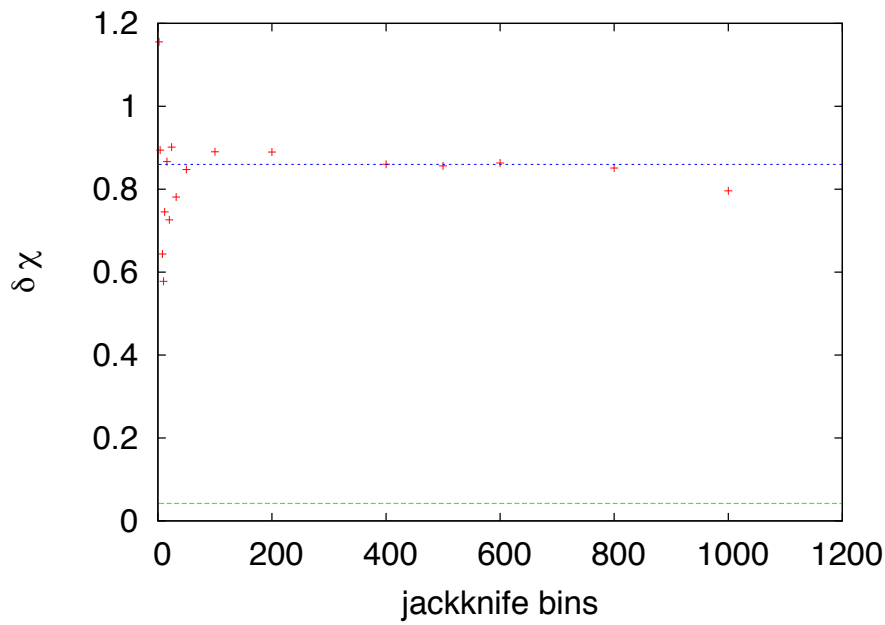



Figure 13.29: Figure 13.28 magnified in the region of the plateau in the values of $\delta\chi$. Convergence is observed for $100 < n_b < 800$ to $\delta\chi = 0.86$. The values $\delta\chi$ and $\delta_c\chi$ are shown in the plots by the two horizontal lines. The ratio $\delta\chi/\delta_c\chi \approx \sqrt{1 + 2\tau_m}$.

## 13.9 Problems

1. Prove that equation (13.22) satisfies the detailed balance condition.

2. Write a program that prints the memory used by variables of the type `character`, `integer`, `integer(8)`, `real`, `real(8)` in bytes. Calculate the amount of memory needed for an array of size 2,000,000 for each of the above types of variables.

3. Make the appropriate changes in the Ising model program so that it measures the average acceptance ratio $\bar{A}$ of the Metropolis steps. I.e. compute the ratio of accepted spin flips to the number of attempted spin flips. Compute the dependence of $\bar{A}$ on the temperature and the size of the system. Take $L = 20$ and $\beta = 0.20, 0.30, 0.40, 0.42, 0.44, 0.46, 0.48, 0.50$. Then take $\beta = 0.20$, $L = 10, 20, 40, 80, 100$. Repeat for the same values of $L$ for $\beta = 0.44$ and $\beta = 0.48$.

4. Reproduce the plots in figure 13.12 and compute $\tau_m$. Repeat for $\tau_e$. Compare your results with $\tau_{\text{int,m}}$ and $\tau_{\text{int,e}}$.

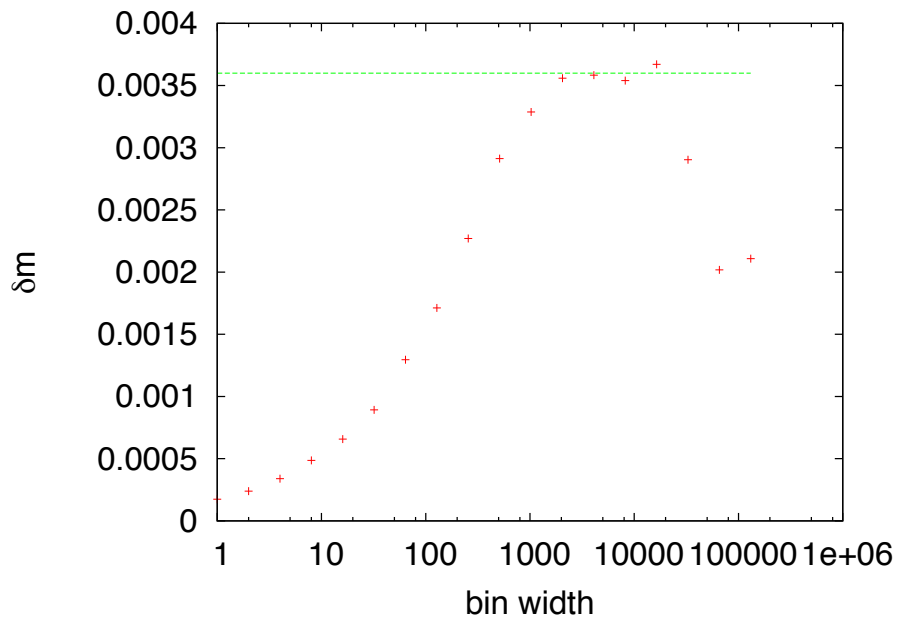



Figure 13.30: The error $\delta m$ calculated using the bootstrap method. We take $n_S = 1000$ and split our data into bins whose bin width is shown on the horizontal axis. Each bin is considered as an independent measurement. For (bin width) = 1 we obtain $\delta_c m = 0.00017$. When (bin width)$> 2\tau_m$ the error becomes $\delta m = 0.0036$. This is read off from the plateau for $1100 <$ (bin width) $< 16000$. The result is consistent with the one obtained using the jackknife method. The horizontal lines are $\delta_c m$ and $\delta m$.

5. Reproduce the plots in figure 13.15 and repeat your calculation for the energy.

6. Reproduce the plots in figure 13.17. Repeat your calculation for the energy. Then, construct similar plots for $\tau_{\text{int,m}}$ and $\tau_{\text{int,e}}$ as a function of $t_{\text{max}}$ (see figure 13.14).

7. Reproduce the plots in figures 13.19 and 13.20. Repeat your calculation for the energy. Then, construct similar plots for $\tau_{\text{int,m}}$ and $\tau_{\text{int,e}}$ as a function of $t_{\text{max}}$ (see figure 13.14).

8. Modify the Ising model program presented in the text so that it can simulate the Ising model in the presence of an external magnetic field $B$ (see equation (13.2)). Calculate the magnetization per site $m(\beta, B)$ for $L = 32$ and $B = 0.2, 0.4, 0.6, 0.8, 1.0$ at an interesting range of temperatures. Use different initial configuration in order to study the thermalization of the system as $B$ increases: Cold state

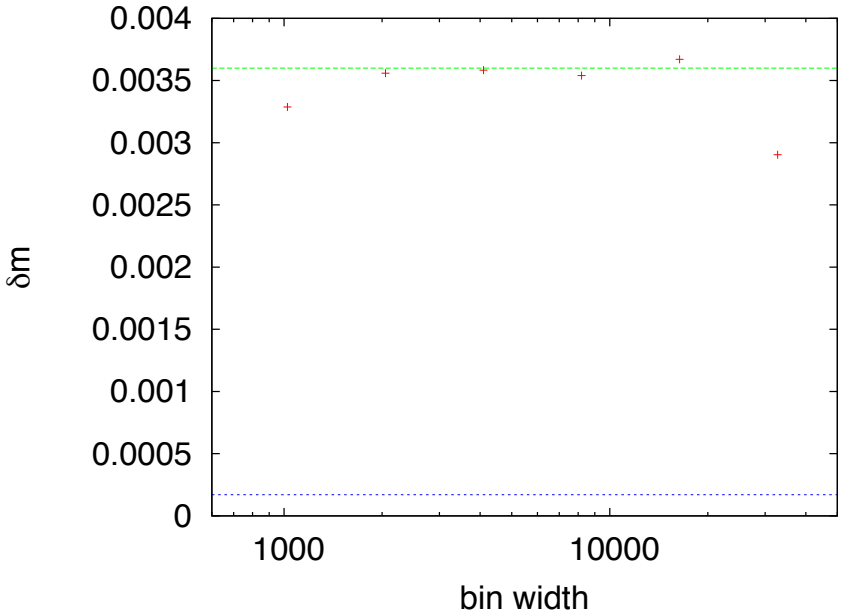



Figure 13.31: The plot of figure 13.30 near the plateau in the values of $\delta m$.

with spins parallel to $B$, cold state with spins antiparallel to $B$ and hot state. Study the dependence of the critical temperature separating the ordered from the disordered state on the value of $B$.

9. Hysteresis: In the previous problem, the Ising model with $B \neq 0$ has a first order phase transition, i.e. a discontinuity in the value of the order parameter which in our case is the magnetization as a function of $B$. Near a first order transition we observe the phenomenon of hysteresis. In order to see it, set $L = 32$ and $\beta = 0.55$ and

   (a) thermalize the system for $B = 0$

   (b) simulate the system for $B = 0.2$ using as an initial state the last one coming from the previous step. Do 100 sweeps and calculate $\langle m \rangle$.

   (c) continue by increasing each time the magnetic field by $\delta B = 0.2$. Stop when $\langle m \rangle \approx 0.95$.

   (d) using the last configuration from the previous step, repeat by decreasing the magnetic field by $\delta B = -0.2$ until $\langle m \rangle \approx -0.95$.

   (e) using the last configuration from the previous step, repeat by increasing the magnetic field by $\delta B = 0.2$ until $\langle m \rangle \approx 0.95$.

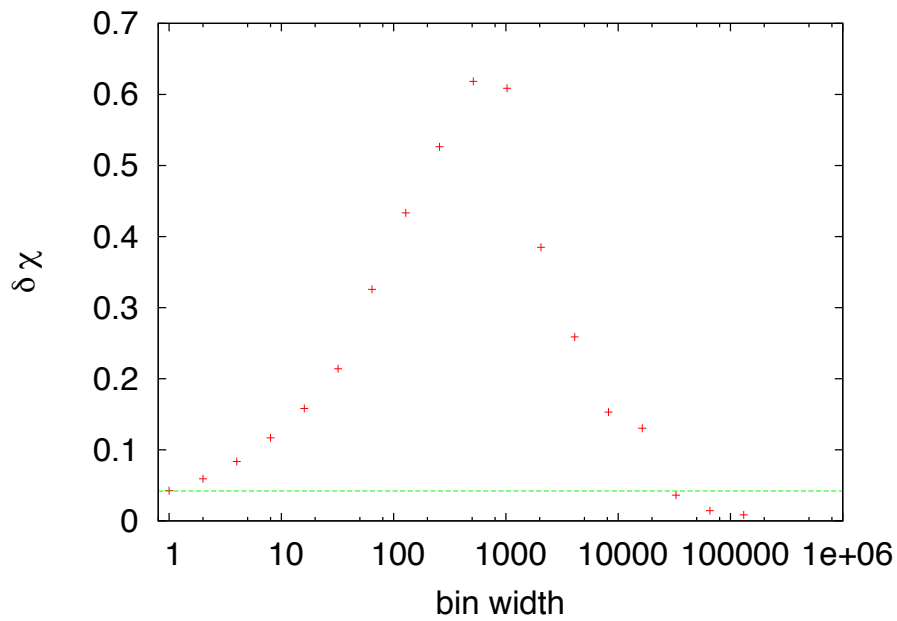



Figure 13.32: The error $\delta\chi$ calculated using the bootstrap method. We take $n_S = 1000$ and split our data into bins whose bin width is shown on the horizontal axis. Each bin is considered as an independent measurement. For (bin width) = 1 we obtain $\delta_c\chi = 0.0421$. When (bin width) $> 2\tau_m$ the error becomes $\delta\chi = 0.615$. This is read off from the plateau for $500 <$ (bin width) $< 1000$ (see figure 13.33). The result is consistent with the one obtained using the jackknife method. The horizontal lines are $\delta_c\chi$ and $\delta\chi$.

Make the plot $(B, m)$. What do you observe?

For systems near a first order phase transition, the order parameter can take two different values with almost equal probability. This means that the free energy has two local minima. Only one of them is the true, global minimum. This is depicted in figure 12.2 where two equally probable values of the order parameter are shown. This happens exactly at the critical point. When we move away from the critical point, one of the peaks grows and it is favored corresponding to the global minimum of the free energy. The local minimum is called a *metastable state* and when the system is in such a state, it takes a long time until a thermal fluctuation makes it overcome the free energy barrier and find the global minimum. In a Monte Carlo simulation such a case presents a great difficulty in sampling states correctly near the two local minima. Repeat the above simulations, this time making $100,000$ sweeps per point. Plot the time series of

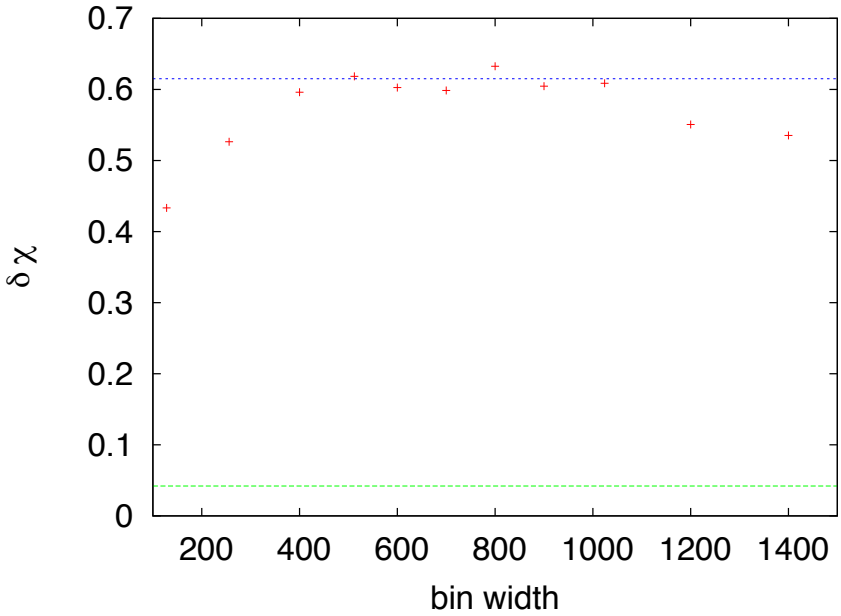



Figure 13.33: The plot of figure 13.32 near the plateau in the values of $\delta\chi$.

the magnetization and observe the transitions from the metastable state to the stable one and backwards. Compute the histogram of the values of the magnetization and determine which state is the metastable in each case. How is the histogram changing as $B$ is increased?

10. Write a program that simulates the 2 dimensional Ising model on a *triangular* lattice using the Metropolis algorithm. The main difference is that the number of nearest neighbors is $z = 6$ instead of $z = 4$. Look into chapter 13.1.2 of Newman and Barkema (esp. figure 13.4). Compute the change in energy for each spin flip for the Metropolis step. Calculate the maxima of the magnetic susceptibility and of the specific heat and see if they are close to the expected critical temperature $\beta_c \approx 0.274653072$. Note that even though $\beta_c$ is different than the corresponding value on the square lattice, the critical exponents are the same due to universality.

11. Write a program that simulates the *three* dimensional Ising model on a cubic lattice using the Metropolis algorithm. Use helical boundary conditions (all you need in this case is to add a parameter `ZNN=L*L` together with the `XNN=1` and `YNN=L`).

12. Write a program that simulates the three dimensional Ising model on a cubic lattice using the Metropolis algorithm. Use *periodic* boundary conditions. (Hint: Use a one dimensional array `s(N)`. During initialization, compute the arrays `XNN(-N:N)`, `YNN(-N:N)`, `ZNN(-N:N)` which store the nearest neighbors of the position `i` on the lattice in `XNN(i)`, `XNN(-i)`, `YNN(i)`, `YNN(-i)`, `ZNN(i)`, `ZNN(-i)`.)

13. Simulate the *antiferromagnetic* two dimensional Ising model on a square lattice using the Metropolis algorithm. You may use the same code that you have and enter negative temperatures. Find the ground state(s) of the system.

    Define the *staggered magnetization* $m_s$ to be the magnetization per site of the sublattice consisting of sites with odd $x$ *and* $y$ coordinate. Set $L = 32$ and compute the energy, the $m_s$, the specific heat, the magnetic susceptibility $\chi$ and the *staggered* magnetic susceptibility $\chi_s = \beta N/4\langle(m_s - \langle m_s\rangle)^2\rangle$.

    $\chi$ has a maximum in the region $\beta \approx 0.4407$. Compute its value at this temperature for $L = 32 - 120$. Show that $\chi$ does not diverge as $L \to \infty$, therefore $\chi$ does not show a phase transition.

    Repeat the calculation for $\chi_s$. What do you conclude? Compare the behavior of $\langle m_s\rangle$ for the antiferromagnetic Ising model with $\langle m\rangle$ of the ferromagnetic.

14. Modify the program in `boot.f90` so that it bins its input data. Reproduce the plots in figures 13.30 and 13.32. (Hint: See the file `boot_bin.f90`)

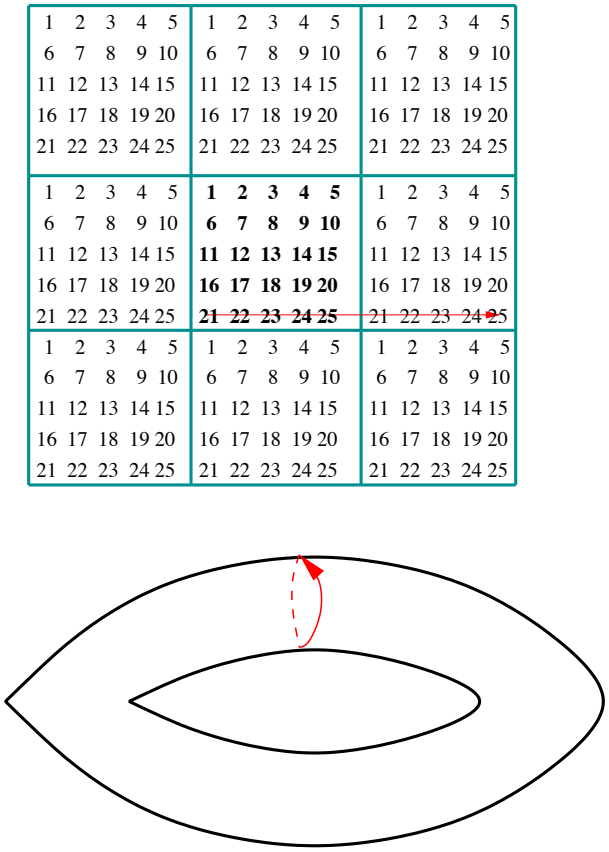


Figure 13.34: Horizontal motion on the $L = 5$ square lattice with periodic boundary conditions. The trajectory is a circle.

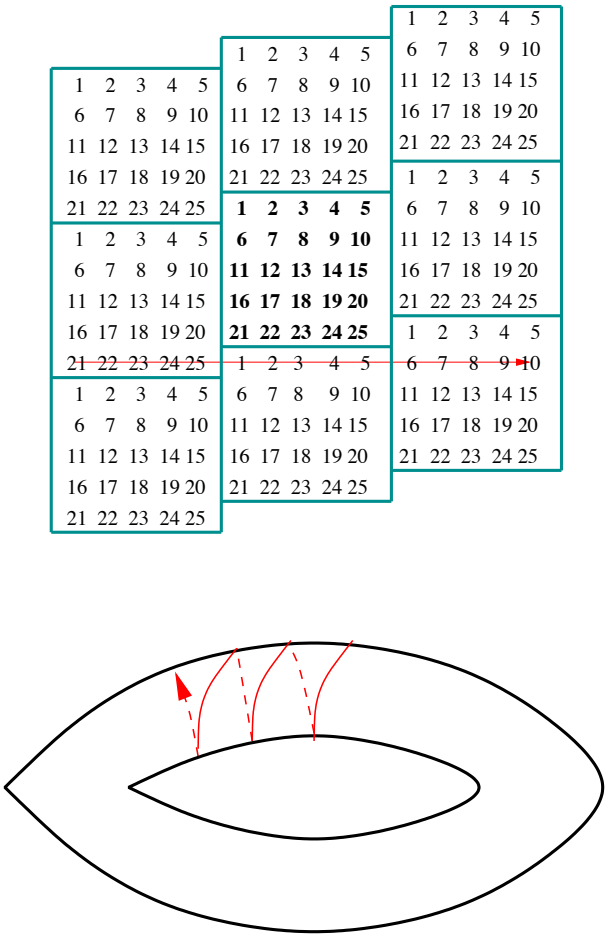


Figure 13.35: Horizontal motion on the $L = 5$ square lattice with helical boundary conditions. The trajectory is a spiral.

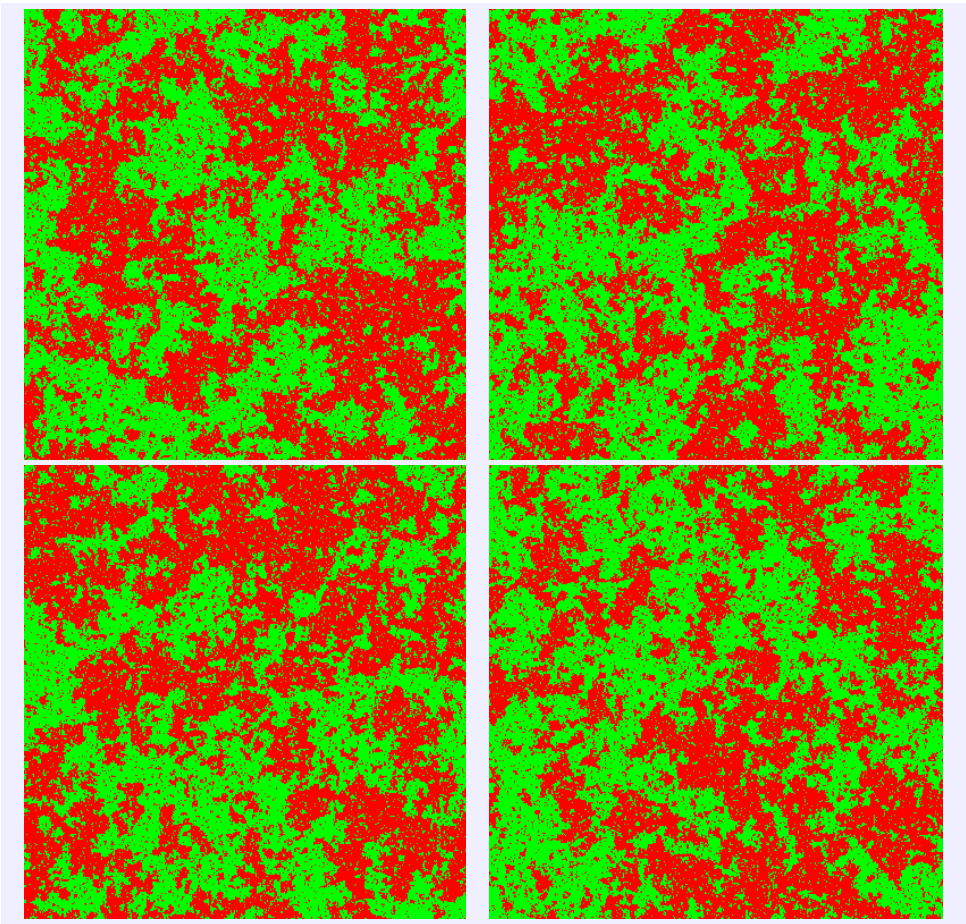


Figure 13.36: Spin configurations for the Ising model with $L = 400$, $\beta = 0.4292$ after 4000, 9000, 12000 και 45000 sweeps respectively. We observe the formation of large clusters of same spin. This makes hard to form a new independent configuration with the Metropolis algorithm and results in large autocorrelation times.

# Chapter 14

# Critical Exponents

In the previous chapters, we saw that when a system undergoes a continuous phase transition as $\beta \to \beta_c$, or equivalently as the reduced temperature[1]

$$t \equiv \frac{\beta_c - \beta}{\beta_c} \to 0\,, \tag{14.1}$$

the correlation length $\xi \equiv \xi(\beta, L = \infty)$, calculated in *the thermodynamic limit* diverges according to the relation

$$\xi \sim |t|^{-\nu} \qquad (\nu = 1 \text{ for 2d-Ising})\,. \tag{14.2}$$

The behavior of such systems near the phase transition is characterized by *critical exponents*, such as the exponent $\nu$, which are the same for all systems in the same *universality class*. The critical exponents describe the leading non analytic behavior of the observables in the thermodynamic limit[2] $L \to \infty$, when $t \to 0$. Systems with the same long distance behavior, but which could possibly differ microscopically, belong to the same *universality class*. For example, if we add a next to nearest neighbor interaction in the Hamiltonian of the Ising model or if we consider the system on a triangular instead of a square lattice, the system will still belong to the same universality class. As $\xi \to \infty$ these details become irrelevant and all these systems have the same long distance behavior. Microscopic degrees of freedom of systems in the same universality class can be quite different, as is the case of the liquid/vapor phase transition at the triple point and the Ising model.

[1]You may also find the definition $t = (T - T_c)/T_c$ but as $t \ll 1$ the two definitions are almost equivalent (they differ by a term of order $\sim t^2$).

[2]Beware: We *first* take $L \to \infty$ and *then* $t \to 0$.

The critical exponents of the 2d Ising model universality class are the Onsager exponents:

$$\chi \sim |t|^{-\gamma}\,, \qquad \gamma = 7/4\,, \tag{14.3}$$

$$c \sim |t|^{-\alpha}\,, \qquad \alpha = 0 \qquad \text{and} \tag{14.4}$$

$$\langle m \rangle \sim |t|^{\beta} \qquad t < 0\,, \qquad \beta = 1/8\,. \tag{14.5}$$

This behavior is seen only in the thermodynamic limit $L \to \infty$. For a finite lattice, all observables are analytic since they are calculated from the analytic[3] partition function $Z(\beta)$ given by equation (13.4). When $1 \ll \xi \ll L$ the model behaves approximately as the infinite system. As $\beta \approx \beta_c$ and $\xi \sim L$ *finite size effects* dominate. Then the fluctuations, e.g. $\chi$ and $c$, on the finite lattice have a maximum for a *pseudocritical temperature* $\beta_c(L)$ for which we have that[4]

$$\lim_{L\to\infty} \beta_c(L) = \beta_c\,. \tag{14.6}$$

For the Ising model on the square lattice, defined by (13.14), we have that $\beta_c = \log(1+\sqrt{2})/2$.

Because of (14.2), when on the finite lattice we take $\beta = \beta_c(L)$, we have that $\xi(t,L) \sim L \Rightarrow |t| = |(\beta_c - \beta_c(L))/\beta_c| \sim L^{-1/\nu}$, therefore equations (14.3)–(14.5) become

$$\chi \sim L^{\gamma/\nu}\,, \tag{14.7}$$

$$c \sim L^{\alpha/\nu}\,, \tag{14.8}$$

$$m \sim L^{-\beta/\nu}\,. \tag{14.9}$$

The left hand sides of the above relations are normally evaluated at $\beta = \beta_c(L)$, but they can also be evaluated at any temperature in the *pseudocritical region*. Most of the times, one calculates the observables for $\beta = \beta_c(L)$, but one can also use e.g. $\beta = \beta_c$[5]. In the next sections we will show how to calculate the critical exponents by using the scaling relations (14.3)–(14.5) and (14.7)–(14.9).

[3]It is a finite sum of analytic functions, therefore an analytic function.

[4]Each observable may have a slightly different pseudocritical temperature, so we may write $\beta_c^{\chi}(L)$, $\beta_c^{c}(L)$ etc.

[5]In the limit $L \to \infty$ the difference is not important, it comes from analytic terms which don't contribute to the non analytic behavior. In practice, the speed of convergence to the asymptotic behavior may differ.

## 14.1 Critical Slowing Down

The computation of critical exponents is quite involved and requires accurate measurements, as well as simulations of large systems in order to reduce finite size effects. The Metropolis algorithm suffers from severe *critical slowing down*, i.e. diverging autocorrelation times with large dynamic exponent $z$ according to (13.36), near the critical region, which makes it impossible to study large systems. In this section we will discuss the cause of this effect whose understanding will lead us to new algorithms that beat critical slowing down. These are the *cluster algorithms* and, in particular, the *Wolff algorithm*. The success of these algorithms is based on the dynamics of the system and, therefore, they have a more specialized range of applications. In contrast, the Metropolis algorithm can, in principle, be applied on any system studied with the Monte Carlo method.

According to the discussion in section 13.5, the Ising model simulation using the Metropolis algorithm near the critical region exhibits an increase in autocorrelation times given by the scaling relation (13.36)

$$\tau \sim \xi^z \,. \tag{14.10}$$

The correlation length of the finite system becomes $\xi \sim L$ in this region, and we obtain equation (13.36), $\tau \sim L^z$. When $z > 0$ we have the effect of critical slowing down.

Critical slowing down is the main reason that prohibits the simulation of very large systems, at least as far as CPU time $t_{\text{CPU}}$ is concerned[6]. The generation of a given number of configuration requires an effort $t_{\text{CPU}} \sim L^d$. But the measurement of a local quantity, like $\langle m \rangle$, for a given number of times requires no extra cost, since each configuration yields $L^d$ measurements[7]. In this case, measuring for the largest possible $L$ is preferable, since it reduces finite size effects. We see that, in the absence of critical slowing down, the cost of measurement of $\langle m \rangle$ is $t_{\text{CPU}}^{\langle m \rangle} \sim L^0$.

Critical slowing down, however, adds to the cost of production of independent configurations and we obtain $t_{\text{CPU}}^{\langle m \rangle} \sim L^z$, making the large $L$ simulations prohibitively expensive. For the Metropolis algorithm on the two dimensional Ising model we have that $z \approx 2.17$; and the problem is severe. Therefore, it is important to invent new algorithms that beat critical slowing down. In the case of the Ising model and similar

[6]Of course the amount of available memory can be another inhibiting factor.

[7]Each site contributes one measurement!

spin systems, the solution is relatively easy. It is special to the specific dynamics of spin systems and does not have a universal application.

The reason for the appearance of critical slowing down is the divergence of the correlation length $\xi$. As we approach the critical temperature $\beta \to \beta_c$ from the disordered phase, the typical configurations are dominated by large clusters of same spins. The Metropolis algorithm makes at most one spin flip per step and the acceptance ratios for spins inside a cluster are small. For example, a spin with four same neighboring spins can flip with probability $\mathrm{e}^{-8\beta_c} \approx 0.029$, which is quite small. The spins that change more often are the ones with more neighbors having opposite spins, therefore the largest activity is observed at the boundaries of the large clusters. In order to obtain a statistically independent configuration, we need to destroy and create many clusters, something that happens very slowly using the Metropolis algorithm who realizes this process mostly by moving the boundaries of the clusters.

## 14.2 Wolff Cluster Algorithm

Beating critical slowing down requires new algorithms so that at each step a spin configuration is changed at the scale of a spin cluster[8]. The cluster algorithms construct such regions of same spins in a way that the proposed new configuration has all the spins of the clusters flipped. For such an algorithm to be successful, the acceptance ratios should be large. The most famous ones are the Swendsen-Wang [60] and the Wolff [61] cluster algorithms.

The process of constructing the clusters is stochastic and depends on the temperature. Small clusters should be favored for $\beta \ll \beta_c$, whereas large clusters of size $\sim L$ should dominate for $\beta \gg \beta_c$.

The basic idea of the Wolff algorithm is to choose a site randomly, a so called *seed* of the cluster, and construct a spin cluster around it. At each step, we add new members to the cluster with probability $P_{\text{add}} = P_{\text{add}}(\beta)$. If $P_{\text{add}}(\beta)$ is properly chosen, the detailed balance condition (12.59) is satisfied and the new configuration is *always* accepted. This process is depicted in figure 14.1. In the state $\mu$, the cluster is enclosed by the dashed line. The new state $\nu$ is obtained by flipping all the spins in the cluster, leaving the rest of the spins to be the same.

[8] A spin cluster is a subset of the lattice composed of connected lattice sites of same spins.

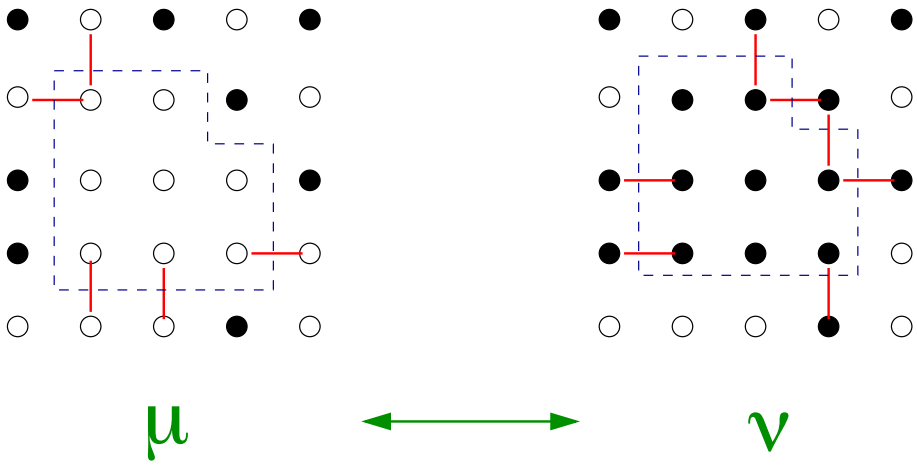



Figure 14.1: Two spin configurations that differ by the flip of a *Wolff cluster*. The bonds that are destroyed/created during the transition belong to the boundary of the cluster.

The correct choice of $P_{\text{add}}$ will yield equation (12.60)

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \mathrm{e}^{-\beta(E_\nu - E_\mu)} \,. \tag{14.11}$$

The discussion that follows proves (14.11) and can be found in the book by Newman and Barkema [4]. The crucial observation is that the change in energy in the exponent of the right hand side of (14.11) is due to the creation/destruction of bonds on the *boundary* of the cluster. The structure of the bonds in the interior of the cluster is identical in the two configurations $\mu$ and $\nu$. This can be seen in the simple example of figure 14.1. By properly choosing the selection probability $g(\mu \to \nu)$ of the new state $\nu$ and the acceptance ratio $A(\mu \to \nu)$, so that

$$P(\mu \to \nu) = g(\mu \to \nu)\, A(\mu \to \nu) \,, \tag{14.12}$$

we will succeed in satisfying (14.11) and maximize the acceptance ratio. In fact in our case we will find that $A(\mu \to \nu) = 1$!

The selection probability $g(\mu \to \nu)$ is the probability of constructing a particular cluster and can be split in three factors:

$$g(\mu \to \nu) = p_{\text{seed}} \times p_{\text{yes}}^{\text{int}} \times p_{\text{no}}^{\text{border}} \,. \tag{14.13}$$

The first term is the probability to start the cluster from the particular seed. By choosing a lattice site with equal probability we obtain

$$p_{\text{seed}} = \frac{1}{N} \,. \tag{14.14}$$

Then the cluster starts growing around its seed. The second term $p_{\text{yes}}^{\text{int}}$ is the probability to include all cluster members found in the *interior* of the cluster. This probability is complicated and depends on the size and shape of the cluster. Fortunately, it is not important to calculate it. The reason is that in the opposite transition $\nu \to \mu$, the corresponding term is exactly the same since the two clusters are exactly the same (the only differ by the value of the spin)!

$$p_{\text{yes}}^{\text{int}}(\mu \to \nu) = p_{\text{yes}}^{\text{int}}(\nu \to \mu) \equiv C_{\mu\nu} \,. \tag{14.15}$$

The third term is the most interesting one. The cluster stops growing when we are on the boundary and say "no" to including all nearest neighbors with same spins, which are not already in the cluster (obviously, the opposite spins are not included). If $P_{\text{add}}$ is the probability to include a nearest neighbor of same spin to the cluster, the probability of saying "no" is $1 - P_{\text{add}}$. Assume that we have $m$ "bonds"[9] of same spins on the boundary of the cluster in the state $\mu$, and that we have $n$ such bonds in the state $\nu$. In figure 14.1, for example, we have that $m = 5$ and $n = 7$. Therefore, the probability to stop the cluster in the state $\mu$ is to say "no" $m$ times, which happens with probability $(1 - P_{\text{add}})^m$:

$$p_{\text{no}}^{\text{border}}(\mu \to \nu) = (1 - P_{\text{add}})^m \,. \tag{14.16}$$

Similarly, the cluster in the state $\nu$ stops at the same boundary with probability

$$p_{\text{no}}^{\text{border}}(\nu \to \mu) = (1 - P_{\text{add}})^n \,. \tag{14.17}$$

Therefore

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \frac{\frac{1}{N}\, C_{\mu\nu}\, (1 - P_{\text{add}})^m\, A(\mu \to \nu)}{\frac{1}{N}\, C_{\mu\nu}\, (1 - P_{\text{add}})^n\, A(\nu \to \mu)} = \mathrm{e}^{-\beta(E_\nu - E_\mu)} \,. \tag{14.18}$$

The right hand side of the above equation depends only on the number of bonds on the boundary of the cluster. The energy difference depends only on the creation/destruction of bonds on the boundary of the cluster and the internal bonds don't make any contribution to it. Each bond created during the transition $\mu \to \nu$ decreases the energy by 2 and each bond destroyed increases the energy by 2:

$$E_\nu - E_\mu = (-2n) - (-2m) = 2(m - n) \,, \tag{14.19}$$

[9] A link with same spins on both sides.

which yields

$$(1-P_{\text{add}})^{m-n}\frac{A(\mu\to\nu)}{A(\nu\to\mu)} = \mathrm{e}^{-2\beta(m-n)} \Rightarrow \frac{A(\mu\to\nu)}{A(\nu\to\mu)} = \left[\mathrm{e}^{2\beta}(1-P_{\text{add}})\right]^{n-m} . \tag{14.20}$$

From the above relation we see that if we choose

$$1-P_{\text{add}} = \mathrm{e}^{-2\beta} \Rightarrow P_{\text{add}} = 1-\mathrm{e}^{-2\beta} , \tag{14.21}$$

then we can also choose

$$A(\mu\to\nu) = A(\nu\to\mu) = 1\,! \tag{14.22}$$

Therefore, we can make the condition (14.11) to hold by constructing a cluster using the $P_{\text{add}}$ given by (14.21), flipping its spins, and *always* accepting the resulting configuration as the new state.

Summarizing, the algorithm for the construction of a Wolff cluster consists of the following steps:

1. Choose a seed by picking a lattice site with probability $p_{\text{seed}} = \frac{1}{N}$. This is the first new member of the cluster

2. Repeat: For each new member of the cluster, visit its nearest neighbors that do not already belong to the cluster. If they have the same spin, add them to the "new members" of the cluster with probability $P_{\text{add}}$. The original spin is not a "new member" anymore

3. When there are no more "new members", the construction of the cluster ends

4. Flip the spin of all the members of the cluster.

The algorithm is ergodic, since every state can be obtained from any other state by constructing a series of clusters of size 1 (equivalent to single flips).

The probability $P_{\text{add}}$ depends on the temperature $\beta$. It is quite small for $\beta \ll \beta_c$ and almost 1 for $\beta \gg \beta_c$. Therefore, in the first case the algorithm favors very small clusters (they are of size 1 for $\beta = 0$) and in the second case it favors large clusters. In the high temperature regime, we have almost random spin flips, like in the Metropolis algorithm. In the low temperature regime, we have large probability of flipping the dominant cluster of the lattice. This is clearly seen in figure 14.2, where the fraction of the average cluster size to the lattice size $\langle n\rangle/N$ is plotted

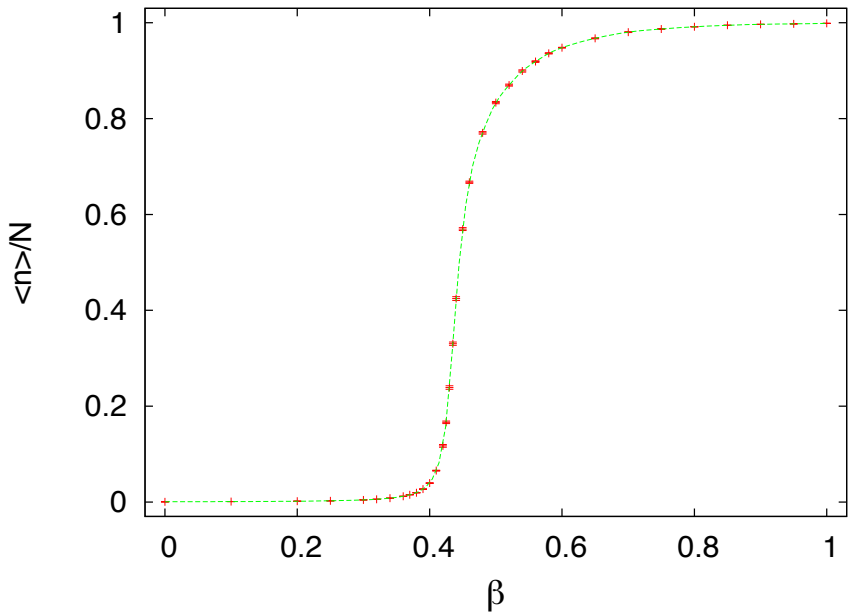



Figure 14.2: The Wolf cluster size as a function of the temperature. The plot shows the average cluster size as a fraction of the lattice size $N$. In the high temperature regime, $\beta \ll \beta_c$, this is $\sim 1/N$, and in the low temperature regime, $\beta \gg \beta_c$, it becomes $\sim 1$. The data is for the Ising model on the square lattice for $L = 40$.

as a function of the temperature. For small $\beta$, $\langle n \rangle / N \to 1/N$ whereas for large $\beta$, $\langle n \rangle / N \to 1$.

Figure 14.3 shows typical spin configurations in the high and low temperature regimes. For small $\beta$, most of the time the algorithm chooses a lattice site randomly and constructs a small cluster around it and flips its spins. The Metropolis algorithm picks a lattice site randomly and flips it most of the times. In both cases, the two algorithms function almost the same way and construct the high temperature disordered spin configurations. For large $\beta$, a typical spin configuration is a "frozen" one: A large cluster of same spins with a few isolated thermal fluctuations of different spins. Most of the times, the Wolff algorithm picks a seed in the dominant cluster and the new cluster is almost the same as the dominant cluster: Most of its sites are included with few ones excluded, which upon flipping of the spins, they will form the new thermal fluctuations. After the flips, the old thermal fluctuations have the same spin as the dominant cluster and they become part of the new dominant cluster. The Metropolis algorithm picks lattice sites randomly: When they belong to the dominant cluster they are seldomly flipped, whereas

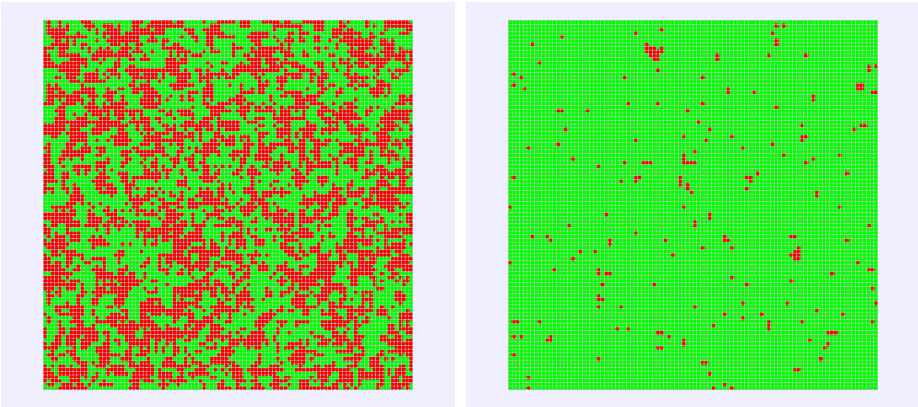


Figure 14.3: A typical spin configuration in the disordered phase (left, $\beta = 0.25$) and in the ordered phase (right, $\beta = 0.5556$) for the Ising model on the square lattice for $L = 100$.

the thermal fluctuations are flipped most of the time. Both algorithms function similarly and have the same efficiency.

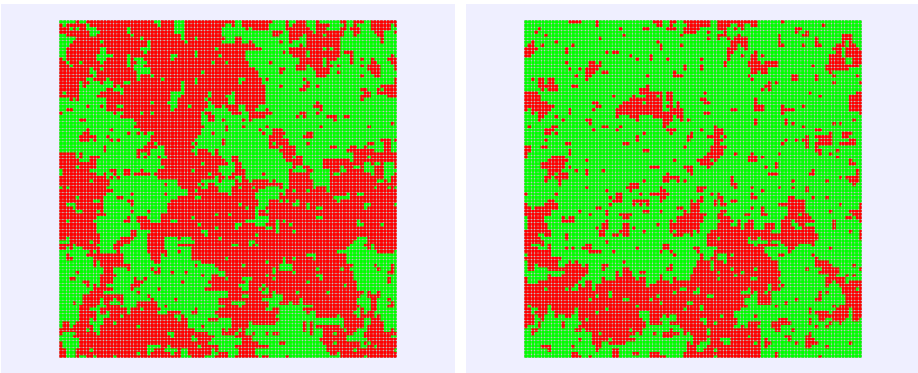


Figure 14.4: Two typical spin configurations in the (pseudo)critical region ($\beta = 0.4348$) for the Ising model on the square lattice for $L = 100$. The two configurations differ by 5000 Metropolis steps.

Figure 14.4 shows typical spin configurations in the critical region. These are dominated by large clusters whose size, shape and position are random. The Wolff algorithm constructs large clusters easily, therefore, large clusters are easily created and destroyed in a few steps (figure 14.2 shows that $\langle n \rangle / N \approx 0.5$). In contrast, the Metropolis algorithm modifies clusters by slowly moving their boundaries and large clusters are destroyed/created very slowly. Autocorrelation times are expected to

reduce drastically when using the Wolff algorithm in the critical region.

The expectation value of the size of the Wolff clusters is a dynamical quantity. In order to see this, we will show that in the disordered phase ($\beta < \beta_c$) we have that

$$\chi = \beta \langle n \rangle \,. \tag{14.23}$$

We take the discussion from Newmann and Barkema [4]: Create a bond on each link of the lattice connecting two same spins with probability $P_{\text{add}} = 1 - \mathrm{e}^{-2\beta}$. In the end, the lattice will be divided in $N_c$ Wolff[10] clusters. Each one will consist of $n_i$ sites, whose spin is $S_i$. Choose a lattice site randomly and flip the spins of the cluster it belongs to. Destroy the bonds and repeat the process[11]. The total magnetization is:

$$M = \sum_{i=1}^{N_c} S_i n_i \,, \tag{14.24}$$

and

$$\langle M^2 \rangle = \langle \left( \sum_{i=1}^{N_c} S_i n_i \right) \left( \sum_{j=1}^{N_c} S_j n_j \right) \rangle = \langle \sum_{i \neq j} S_i S_j n_i n_j \rangle + \langle \sum_i S_i^2 n_i^2 \rangle \,. \tag{14.25}$$

The values $S_i = \pm 1$ are equally probable due to the symmetry of the model, therefore the first term vanishes. Since $S_i^2 = 1$, we obtain

$$\langle m^2 \rangle = \frac{1}{N^2} \langle M^2 \rangle = \frac{1}{N^2} \langle \sum_i n_i^2 \rangle \,. \tag{14.26}$$

In the Wolff algorithm, the creation of a cluster is equivalent to the choice of one of the clusters we created by following the procedure described above. The probability of selecting the cluster $i$ is

$$p_i = \frac{n_i}{N} \,, \tag{14.27}$$

therefore the average value of the size of the Wolff clusters will be

$$\langle n \rangle = \langle \sum_i p_i n_i \rangle = \langle \sum_i \frac{n_i}{N} n_i \rangle = N \langle m^2 \rangle \,. \tag{14.28}$$

[10] These are true Wolff clusters since the bonds have been activated with the correct probability. There are of course isolated sites with no activated bonds around them which are Wolff clusters of size 1.

[11] The result is equivalent to performing one step of the Wolff algorithm, not a very efficient one of course...

By using equation (14.26) and the fact that for $\beta < \beta_c$ we have that $\langle m \rangle = 0$[12], therefore

$$\chi = \beta N(\langle m^2 \rangle - \langle m \rangle^2) = \beta \langle n \rangle \,. \tag{14.29}$$

## 14.3 Implementation

In order to create a cluster around a seed, we need a memory buffer for storing the new members of the cluster. We draw cluster sites from this buffer, and examine whether to add their nearest neighbors to the cluster.

There are two data structures that can be used in this job. The first one is the *stack* (or *LIFO*: last in – first out) and the second one is the *queue* (or *FIFO*: first in – first out). They are both one dimensional arrays, the only difference is how we draw data from them. In the case of a stack, we draw the last element that we stored in it. In the case of the queue, we draw the first element that we stored in it.

The stack is implemented as a one dimensional array `stack(0:N-1)`[13] in which we "push" a new value that we want to store and we "pop" one that we want to retrieve. We use an integer `m` as a pointer to the last value that we stored in the position `stack(m-1)`. `m` is also the number of active elements in the stack. In order to push a value `e` into the stack we:

1. check if there exist available positions in the stack (i.e. if `m<N`)
2. set `stack(m) = e`
3. increase `m` by 1.

In order to pop a value and store it in the variable `e` we:

1. check if the stack is non empty (i.e. if `m>0`)
2. reduce `m` by 1
3. set `e = stack(m)`

The queue implementation is different. The data topology is cyclic, as shown in figure 14.5. We use an array `queue(0:N-1)` and *two* integers `m, n` which point at the beginning and at the end of the buffer. The

[12]This is exactly true only in the thermodynamics limit. For a finite lattice of size $N$, the two quantities differ by a factor of $\beta N \langle m \rangle^2 > 0$, which vanishes in the large $N$ limit.

[13]`stack(0:N-1)` defines an array with elements `stack(0)`, `stack(1)`, ..., `stack(N-1)`

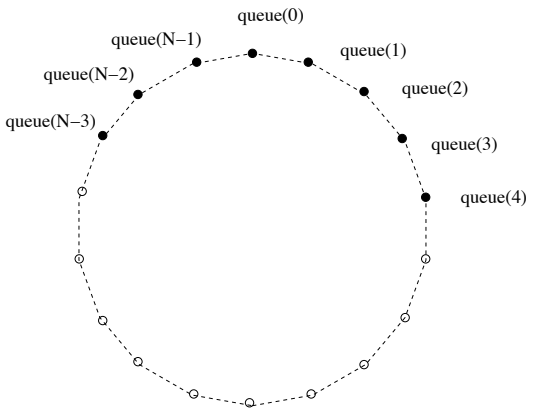



Figure 14.5: Data topology in a queue. In the array depicted here, we have 8 elements stored in `queue(N-3) ... queue(4)`. We have that `m=5, n=N-3, m-n = 8 mod N`. An element is added to the `queue(m)=queue(5)` and an element is popped by calling `queue(n)=queue(N-3)`.

beginning of the data is the element `queue(m-1)` and the end of the data is the element `queue(n)`. When the queue is empty, we have that `m=n` and the same is true when it is full. Therefore we need a flag that flags whether the queue is empty or full. In the beginning we set `flag=0` (queue is empty)[14]. The number `(m-n) mod N` is the number of stored elements[15]. When the queue has data, we set `flag=1`. In order to store a value `e` into the queue we:

1. check whether the queue is full (`m=n` and `flag=1`)
2. set `flag=1`
3. set `queue(m) = e`
4. increase `m` by $1 \mod N$.

In order to pop a value and store it in the variable `e` we:

1. check whether the queue is empty (`m=n` and `flag=0`)
2. set `e = queue(n)`
3. increase `n` by $1 \mod N$

[14]If we choose to store at most `N-1` elements in `queue(N)` then the algorithm becomes slightly simpler (exercise).

[15]Except if `m=n` in which case the number of stored elements is 0 or `N` according to the value of `flag`.

4. if `m=n` set `flag=0`.

Summarizing, the algorithm for constructing a Wolff cluster for the Ising model is the following:

1. choose a seed by randomly picking a site with probability $1/N$
2. check its nearest neighbors. If they have the same spin, add them to the cluster with probability $P_{\text{add}} = 1 - \mathrm{e}^{-2\beta}$. The new members of the cluster are pushed into the stack `stack(0:N-1)` according to the previous discussion
3. pop a site from the stack `stack(0:N-1)`. If the stack is empty we stop the construction and move on to the next step. If not, we check the site's nearest neighbors. If they are not already in the cluster and they have the same spin, we add them to the cluster with probability $P_{\text{add}}$
4. record the size of the cluster and flip the spin of its members.

The choice between stack or queue is not important. The results are the same and the performance similar. The only difference is the way that the clusters are constructed (for the stack, the cluster increases around the seed whereas for the queue it increases first in one direction and then in another). The careful programmer will try both during the debugging phase of the development. Bad random number generators can be revealed in such a test, since the Wolff algorithm turns out to be sensitive to their shortcomings.

### 14.3.1 The Program

The heart of the algorithm is coded in the subroutine `wolff()` in the file `wolff.f90`. Each call to `wolff()` constructs a Wolff cluster, flips its spin and records its size.

The buffer `stack(0:N-1)` is used in order to store the new members of the cluster. We call the function `ALLOCATE` for dynamically allocating the necessary memory and use `DEALLOCATE` before returning to the calling program in order to return this memory back to the system - and avoid memory leaks.

```
ALLOCATE(stack(0:N-1),STAT=chk)
if(chk>0) call locerr('allocation failure for stack in wolff')
 ....
DEALLOCATE(stack)!free memory of stack
```

If the requested memory is not available, then `chk>0` and the subroutine `locerr()` stops the program.

The seed is chosen randomly by a call to `ranlux`:

```
call ranlux(r,1)
cseed    =  INT(N*r)+1
stack(0) =  cseed
nstack   =  1          !the stack has 1 member, the seed
sold     =  s(cseed)
snew     = -s(cseed)   !the new spin value of the cluster
s(cseed) =  snew       !we flip all new members of cluster
ncluster =  1          !size of cluster=1
```

The seed is stored in `cseed` which is immediately added to the cluster (`stack(0)=cseed`). The variable `nstack` records the number of elements in the stack and it is originally set equal to 1. The variable `ncluster` counts the number of sites in the cluster and it is originally set equal to 1. `sold=s(cseed)` is the old value of the spin of the cluster and `snew=-sold` is the new one. The value of the spin of a new member of the cluster is *immediately* changed (`s(cseed)=snew`)! This increases the efficiency of the algorithm. By checking whether the spin of a nearest neighbor is equal to `sold`, we check whether the spin is the same as that of the cluster *and* if it has already been included in the cluster during a previous check.

The loop over the new members of the cluster is summarized below:

```
 do while(nstack > 0)
!pull a site off the stack:
  nstack  = nstack     - 1; scluster = stack(nstack)
!check its four neighbors:
!-------------scluster + XNN:
  nn       =  scluster + XNN; if(nn > N) nn = nn - N
  if(s(nn) == sold)then
   call ranlux(r,1)
   if(r<padd)then
    stack(nstack)=nn; nstack = nstack + 1
    s(nn)        =snew
    ncluster     =ncluster+1
   endif
  endif
! ... check other 3 nearest neighbors ...
 enddo
```

The loop `do while(nstack > 0)` is executed while `nstack>0`, i.e. as long as the stack is not empty and there exist new members in the cluster.

The variable `scluster` is the current site drawn from the stack in order to check its nearest neighbors. The line `nn = scluster + XNN; if(nn > N) nn = nn - N` chooses the nearest neighbor to the right and stores it in the variable `nn`. If the spin `s(nn)` of `nn` is equal to `sold`, then this neighbor has the same spin as that of the cluster *and* it has not already been included to the cluster (otherwise its spin would have been flipped). The variable `padd` is equal to $P_{\text{add}}$(it has been set in `init`) and if `r<padd` (which happens with probability $P_{\text{add}}$), then we add `nn` to the cluster: We add `nn` to the `stack`, we flip its spin (`s(nn)=snew`) and increase the cluster size by 1. We repeat for the rest of the nearest neighbors. The full code is listed below:

```
subroutine wolff
 use global_data
 implicit none
 integer              :: cseed,nstack,sold,snew,scluster,nn,chk
 integer              :: ncluster
 real(8)              :: r
 integer,allocatable  :: stack(:)
!allocate stack memory:
 ALLOCATE(stack(0:N-1),STAT=chk)
 if(chk>0) call locerr('allocation failure for stack in wolff')
!choose a seed for the cluster, put it on the stack and flip it
 call ranlux(r,1)
 cseed    =  INT(N*r)+1
 stack(0) =  cseed
 nstack   =  1          !the stack has 1 member, the seed
 sold     =  s(cseed)
 snew     = -s(cseed)   !the new spin value of the cluster
 s(cseed) =  snew       !we flip all new members of cluster
 ncluster =  1          !size of cluster=1
!start the loop on spins in the stack:
 do while(nstack > 0)
!pull a site off the stack:
  nstack  = nstack     - 1; scluster = stack(nstack)
!check its four neighbors:
!-------------scluster + XNN:
  nn       =  scluster + XNN; if(nn > N) nn = nn - N
  if(s(nn) == sold)then
   call ranlux(r,1)
   if(r<padd)then
    stack(nstack)=nn; nstack = nstack + 1
    s(nn)        =snew
    ncluster     =ncluster+1
   endif
  endif
```

```
!-------------scluster - XNN:
  nn        =  scluster - XNN; if(nn < 1) nn = nn + N
  if(s(nn) == sold)then
   call ranlux(r,1)
   if(r<padd)then
    stack(nstack)=nn; nstack = nstack + 1
    s(nn)        =snew
    ncluster     =ncluster+1
   endif
  endif
!-------------scluster + YNN:
  nn        =  scluster + YNN; if(nn > N) nn = nn - N
  if(s(nn) == sold)then
   call ranlux(r,1)
   if(r<padd)then
    stack(nstack)=nn; nstack = nstack + 1
    s(nn)        =snew
    ncluster     =ncluster+1
   endif
  endif
!-------------scluster - YNN:
  nn        =  scluster - YNN; if(nn < 1) nn = nn + N
  if(s(nn) == sold)then
   call ranlux(r,1)
   if(r<padd)then
    stack(nstack)=nn; nstack = nstack + 1
    s(nn)        =snew
    ncluster     =ncluster+1
   endif
  endif
 enddo !do while(nstack > 0)
 print '(A,I14)','#clu ',ncluster
!-----------------------------------------
 DEALLOCATE(stack)!free memory of stack
!-----------------------------------------
end subroutine wolff
```

In order to link the subroutine with the rest of the program so that we construct one cluster per "sweep"[16], we modify `main()` accordingly:

```
!============= main.f90     ==================
program Ising2D
 use global_data
 implicit none
 integer :: isweep
```

[16]Beware: A Metropolis sweep is not the same as a cluster update. The average size of the cluster changes with $\beta$ and it is quite small for large temperatures (small $\beta$).

```
 call init
 do isweep = 1, nsweep
  if(algorithm .eq. 1)then
   call wolff
  else
   call met
  endif
  call measure
 end do
 call endsim
end program Ising2D
```

The (global) variable `algorithm` controls whether the Wolff[17] or the Metropolis algorithm will be used for the spin updates. The (global) variable padd$\equiv P_{\text{add}} = 1 - \mathrm{e}^{-2\beta}$ is defined in `init()`. The following lines are added: into the file `global_data.f90`

```
 real(8)                 :: padd
 integer                 :: algorithm
```

The following lines are added to the file `init.c`

```
 algorithm=0             !default is metropolis,1 is wolff
 padd      = 1.0D0 - exp(-2.0D0*beta)
```

The following lines are added to the file `options.f90`

```
  .....
  select case(getopt("-hL:b:s:S:n:r:uw"))
  case( 'w' )
   algorithm = 1
  .....
```

in order to add the option `-w` to the command line. This option sets `algorithm=1`, which makes the program run the Wolff algorithm instead of the Metropolis. Some extra info must also be added to the help message printed by `usage` and `simmessage` and ... we are ready! For the compilation we use the `Makefile`

```
FC      =  gfortran
```

[17]Make the appropriate changes so that `wolff()` is called until the clusters constructed will have total size at least equal to `N`.

```
OBJS   =  global_data.o getopt.o main.o init.o met.o wolff.o \
          measure.o end.o options.o ranlux.o
FFLAGS = -O2

is: $(OBJS)
    $(FC) $(FFLAGS) $^ -o $@

$(OBJS):   global_data.f90
options.o: getopt.f90
%.o: %.f90
    $(FC) $(FFLAGS)   -c -o $@ $<
```

The commands

```
> make
> ./is -h
Usage: ./is [options]
       -L: Lattice length (N=L*L)
       -b: beta  (options beta overrides the one in config)
       -s: start (0 cold, 1 hot, 2 old config.)
       -S: seed  (options seed overrides the one in config)
       -n: number of sweeps and measurements of E and M
       -w: use wolff algorithm for the updates
       .......
> ./is -L 20 -b 0.44 -s 1 -S 34235322 -n 5000 -w > outL20b0.44
```

do the compilation, print the usage instructions of the program and perform a test run for $L = 40$, $\beta = 0.44$, by constructing 5000 clusters, starting from a hot configuration and writing the data to the file `outL20b0.44`.

## 14.4 Production

In order to study the Ising model on a square lattice of given size $N$, we have to perform simulations for many values of $\beta$. Then, we want to study the finite size properties and extrapolate the results to the thermodynamic limit, by repeating the process for several values of $N$. The process is long and ... boring. Moreover, a bored researcher makes mistakes and several bugs can enter into her calculations. Laziness is a virtue in this case and it is worth the trouble and the time investment in order to learn some techniques that will make our life easier, our work more efficient, and our results more reliable. Shell scripting can be used in order to code repeated tasks of the command line. In its simplest form, it is just a series of commands written into a text file. Such an example can be found in the file `run1`:

```
# ################### run1 ########################
./is -L 20 -b 0.10 -s 1 -n 5000 -w -S 3423 > outL20b0.10
./is -L 20 -b 0.20 -s 2 -n 5000 -w > outL20b0.20
./is -L 20 -b 0.30 -s 2 -n 5000 -w > outL20b0.30
./is -L 20 -b 0.40 -s 2 -n 5000 -w > outL20b0.40
./is -L 20 -b 0.42 -s 2 -n 5000 -w > outL20b0.42
./is -L 20 -b 0.44 -s 2 -n 5000 -w > outL20b0.44
./is -L 20 -b 0.46 -s 2 -n 5000 -w > outL20b0.46
./is -L 20 -b 0.48 -s 2 -n 5000 -w > outL20b0.48
./is -L 20 -b 0.50 -s 2 -n 5000 -w > outL20b0.50
./is -L 20 -b 0.60 -s 2 -n 5000 -w > outL20b0.60
./is -L 20 -b 0.70 -s 2 -n 5000 -w > outL20b0.70
```

The first line is a comment, since everything after a # is ignored by the shell. The second line starts a simulation from a hot configuration (`-s 1`), lattice size `L=20` (`-L 20`) and temperature $\beta = 0.10$ (`-b 0.10`). The seed for the random number generator is set equal to 3423 (`-S 3423`) and we measure on 5000 Wolff clusters (`-n 5000 -w`). The results, printed to the `stdout`, are redirected to the file `outL20b0.10` (`> outL20b0.10`).

The next ten lines continue the simulation for $\beta = 0.20 - 0.70$. Each simulation starts from the configuration stored in the file `conf` at the end of the previous simulation.

In order to run these commands, the file `run1` should be given execute permissions (only once, the permissions ... stay after that) using the command `chmod`:

```
> chmod a+x run1
```

Then `run1` can be executed like any other command:

```
> ./run1
```

Not bad... But we can do better! Instead of adding one line for each simulation, we can use the programming capabilities of the shell. Let's see how. The file `run2` contains the commands:

```
#!/bin/tcsh -f
# ################### run2 ################################
set L       = 20
set betas   = (0.10 0.20 0.30 0.40 0.42 0.44 0.46 0.48 0.50\
               0.60 0.70)
set start   = "-s 1 -S 3423"
set nsweeps = 5000
```

```
foreach beta ($betas)
 echo "L= $L beta= $beta"
 ./is -L $L -b $beta -n $nsweeps -w $start > outL${L}b${beta}
 set start = "-s 2"
end
```

The first line[18] calls the shell `tcsh` in order to interpret the script. This was not necessary in `run1`, since every shell can interpret the commands that it contains. But in this case we use syntax which is special to the shell `tcsh`.

The second line is a comment.

The third line defines a shell variable whose name is `L`. Its value is set after the = character equal to the *string* `"20"`. This value is accessible by adding a `$` in front of the name of the variable. Therefore, whenever we write `$L` (or `${L}`), the shell substitutes the string of characters `20`. For example, in place of `outL${L}b` the shell constructs the string `outL20b`.

The fourth line defines an array, whose name is `betas`. The different elements of the array can be accessed by using the syntax `$betas[number]`, where "number" is the array element starting from 1. In the example shown above `$betas[1]= 0.10, $betas[2]= 0.20, ..., $betas[11]= 0.70`. The special variable `$#betas` is the number of elements in the array, which is equal to 11. When we write `$betas`, the shell expands it to *all* the values in the array[19].

The fifth line defines the variable `start` to be equal to the string of characters `"-s 1 -S 3423"`. The quotes have been put because we want it to be treated as a single string of characters. If we omit them, then the shell treats `-s`, `1`, `-S` and `3423` as separate words, and we obtain a syntax error. Everything after the character `#` is a comment.

The command `foreach` is a way to construct a loop in `tcsh`. The commands between the `foreach` and `end` repeat once for every word in the parentheses in the `foreach` line. Each time, the loop variable, whose name is put after the keyword `foreach`, is set equal to the next word in the parenthesis. In our case, these words are the values of the array `betas`, and the loop will execute 11 times, once for each value 0.10, 0.20, ... , 0.70, each time with `$beta` set equal to one of those values.

The next three lines are the commands that are repeated by the `foreach` loop. The command `echo` "echoes" its arguments to the `stdout`

[18]The syntax is very strict and the line has to start *exactly* with the characters `#!` The string following `#!` can be the name of any program in the filesystem, which will be used to interpret the script.

[19]Try the command: `echo $betas[3] $#betas $betas`

and informs us about the current value of the parameters used in the simulation (quite useful, especially when the simulations take a long time). The command `./is` runs the program, each time using a different value of `beta`. Notice that the name of the file in which we redirect the `stdout` changes each time that `beta` changes value. Therefore our data will be stored in the files `outL20b0.10, outL20b0.20, ..., outL20b0.70`. The third command forces the program to read the initial configuration from the file `conf`. The first time that the loop is executed, the value of `start` is `"-s 1 -S 3423"` (hot configuration, seed equal to 3423), whereas for all the next simulations, `start` is equal to `"-s 2"` (old configuration).

We can also include a loop over many values of `L` as follows:

```
#!/bin/tcsh -f

set Ls      = (10 20 40)
set betas   = (0.10 0.20 0.30 0.40 0.42 0.44 0.46 0.48 0.50\
               0.60 0.70)
set nsweeps = 5000

foreach  L    ($Ls   )
 set start   = "-s 1 -S 3423"
 foreach beta ($betas)
  echo "L= $L beta= $beta"
  ./is -L $L -b $beta -n $nsweeps -w $start > outL${L}b${beta}
  set start = "-s 2"
 end
end
```

The array variable `Ls` stores as many `L` as we wish. Note that the definitions of `start` are put in a special place (why?).

## 14.5 Data Analysis

Data production must be monitored by looking at the time histories of properly chosen observables. This will allow us to spot gross mistakes and it will serve as a qualitative check of whether the system has thermalized and how long are the autocorrelation times. It is easy to construct time histories using `gnuplot`. For example, the commands[20]:

```
gnuplot> plot "<grep -v '#'            outL40b0.44" \
```

[20]The `plot` command accepts, in place of the name of a data file, the `stdout` of a command `command` using the syntax `plot "<command"`.

```
                u 1          with lines title "E"
gnuplot> plot "<grep -v '#'             outL40b0.44" \
                u (abs($2)) with lines title "|M|"
gnuplot> plot "<awk '/#clu/{print $2}' outL40b0.44" \
                u 1          with lines title "n"
```

show us the time histories of the energy, of the (absolute value of the) magnetization and of the size of the clusters in a simulation with $L = 40$ and $\beta = 0.44$.

The expectation values of the energy per link $\langle e \rangle = \frac{1}{2N}\langle E \rangle$ and the magnetization per site $\langle m \rangle = \frac{1}{N}\langle M \rangle$ with their errors can be calculated by the jackknife program, which can be found in the file `jack.f90` in the directory `Tools` (see appendix 13.8). We compile the program into an executable file `jack` which we copy into the current working directory. The expectation value $\langle e \rangle$ can be calculated using the command:

```
> grep -v # outL40b0.44 |\
  awk  -v L=40 'NR>500{print $1/(2*L*L)}' | ./jack
```

We pass the value `L=40` to the program `awk` by using the option `-v`, therefore making possible the calculation of the ratio of the first column `$1` by $2N = 2L^2$. The condition `NR>500` makes the printing command to be executed only after `awk` reads the first 500 lines[21]. This way we can discard a number of thermalization sweeps. The result of the above command is printed to the `stdout` as follows:

```
# NDAT = 4500 data. JACK = 10 groups
# <o>, chi= (<o^2>-<o>^2)
# <o> +/- err                    chi +/- err
-0.71091166666 0.0024162628283 0.0015719190590 7.819205433e-05
```

The first three lines are the comments printed by the program `jack`, which inform the user about the important parameters of the analysis. The last line gives $\langle e \rangle$ and its error and then the fluctuations $\langle e^2 \rangle - \langle e \rangle^2$ and their error. The latter must be multiplied by $\beta^2 N$, in order to obtain the specific heat $c$ and its error according to (13.29). By adding a few more lines to the command shown above, this multiplication can be done on the fly:

```
> set L = 40; set b = 0.44 ;                \
  grep -v # outL${L}b${b}                 | \
```

[21] `NR` is the number of lines (number of records) read by `awk` so far.

```
  awk -v L=$L 'NR>500{print $1/(2*L*L)}' | \
  ./jack | grep -v #                     | \
  awk -v L=$L -v b=$b                      \
   '{print "e",L,b,$1,$2,b*b*L*L*$3,b*b*L*L*$4}'
```

Well, why all this fuzz? Notice that all the commands shown above can be given in one single line in the command line (by removing the trailing \ of each line). By recalling the command, it is easy to obtain the results for a different value of $L$ and/or $\beta$, by editing the values of the variables `L` and/or `b`. The result is

```
e 40 0.42 -0.619523333 0.00189807 0.311391 0.0228302
```

i.e. $\langle e \rangle = -0.6195(19)$ and $c = 0.311(23)$.

We can work in a similar way for computing the magnetization. We have to calculate the absolute value of the second column of the `stdout` of the command `./is`, for every line that does not start with a `#`:

```
> set L = 40 ; set b = 0.42 ;                                  \
  grep -v # outL${L}b${b}                                    | \
  awk -v L=$L 'NR>500{m=($2>0)?$2:-$2;print m/(L*L)}' | \
  ./jack | grep -v #                                         | \
  awk -v L=$L -v b=$b                                          \
   '{print "m",L,b,$1,$2,b*L*L*$3,b*L*L*$4}'
```

The absolute value is calculated by the expression `($2>0)?$2:-$2`, and it is stored in the variable `m`, which in turn is printed after being divided by $N = L^2$. The result is

```
m 40 0.44 0.6250527778 0.00900370 21.8345 1.39975
```

which gives $\langle m \rangle = 0.6251(90)$ and $\chi = 21.8(14)$.

Similarly we can calculate $\langle n \rangle / N$:

```
> set L = 40 ; set b = 0.44 ;                  \
  grep '#clu' outL${L}b${b}                  | \
  awk -v L=$L 'NR>500{print $2/(L*L)}' | \
  ./jack | grep -v #                         | \
  awk -v L=$L -v b=$b '{print "n",L,b,$1,$2}'
```

The result is

```
n 40 0.44 0.4257476389 0.01302602
```

which gives $\langle n \rangle / N = 0.426(13)$.

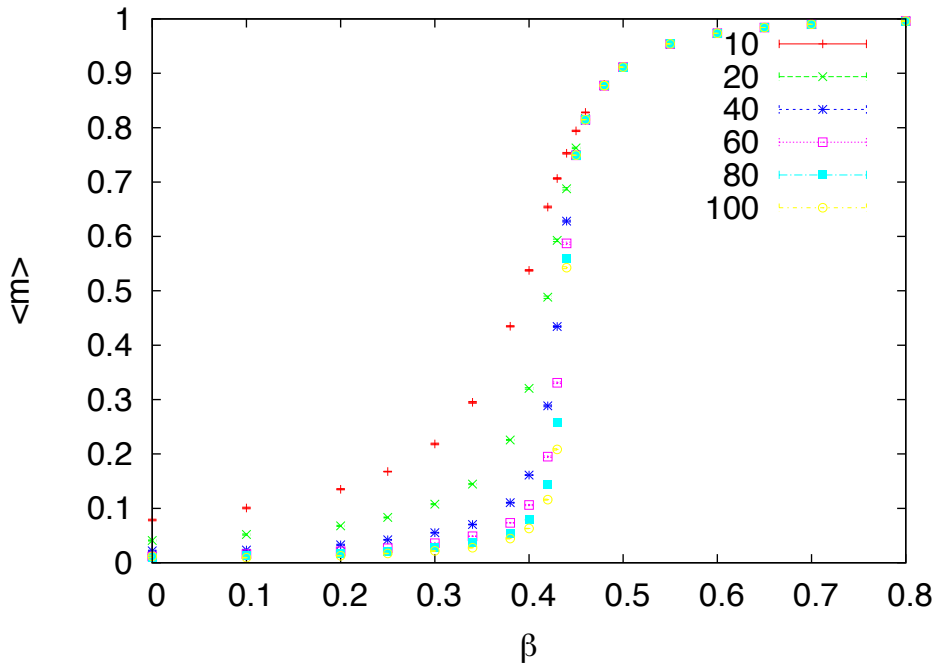



Figure 14.6: The results of the simulations performed by the shell script in the file `run3`. The expectation value of $\langle m \rangle$ is shown to decrease as $1/L$ at high temperatures $\beta \ll \beta_c$.

All of the above commands can be summarized in the script in the file `run3`:

```
#!/bin/tcsh -f
set Ls      = (10 20 40 60 80 100)
set betas   = (0.00 0.10 0.20 0.25 0.30 0.34 0.38 \
               0.40 0.42 0.43 0.44 0.45 0.46 0.48 \
               0.48 0.50 0.55 0.60 0.65 0.70 0.80 )
set nsweeps = 100000

foreach  L    ($Ls    )
 set start    = "-s 1 -S 3423"
 foreach beta ($betas)
  ./is -L $L -b $beta -n $nsweeps -w $start > outL${L}b${beta}
  set start = "-s 2"
  # Calculate <e> = <E>/(2N and c=beta^2*N*(<e^2>-<e>^2):
  grep -v '#' outL${L}b${beta}            | \
  awk -v L=$L 'NR>500{print $1/(2*L*L)}' | \
```

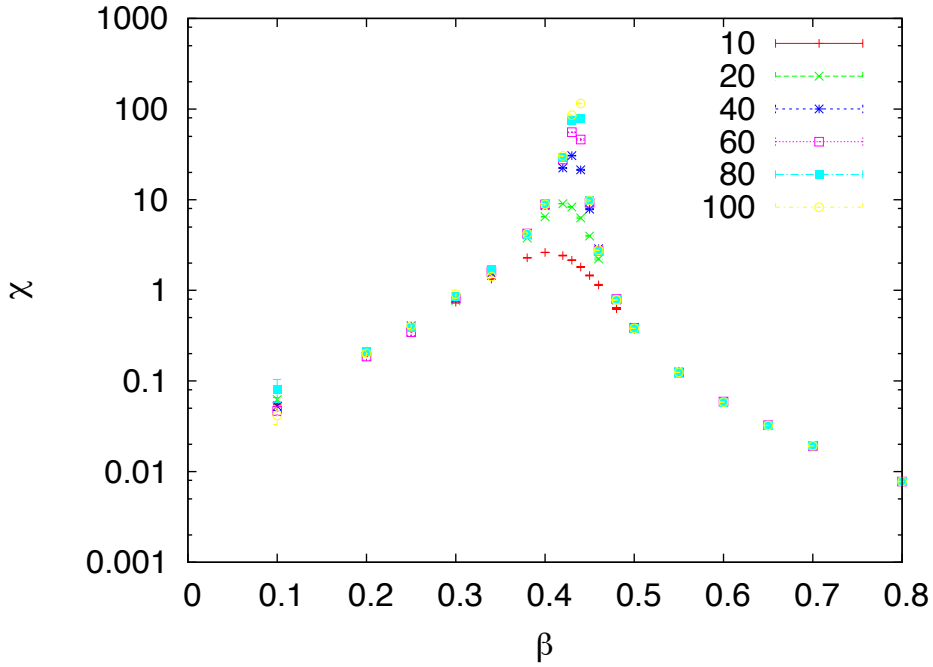



Figure 14.7: The results of the simulations performed by the shell script in the file `run3`. The magnetic susceptibility $\chi$ is shown to be almost independent of the lattice size when $\beta$ takes values away from the critical region. In the critical region, its value increases as shown in equation (13.10).

```
  ./jack | grep -v '#'                    | \
  awk -v L=$L -v b=$beta                    \
   '{print "e",L,b,$1,$2,b*b*L*L*$3,b*b*L*L*$4}'
  # Calculate <m> = <|M|>/N and chi=beta*N*(<m^2>-<m>^2)
  grep -v '#' outL${L}b${beta}                          | \
  awk -v L=$L 'NR>500{m=($2>0)?$2:-$2;print m/(L*L)}' | \
  ./jack | grep -v '#'                                  | \
  awk -v L=$L -v b=$beta                                  \
   '{print "m",L,b,$1,$2,b*L*L*$3,b*L*L*$4}'
 end
end
```

The script is run with the command

```
> ./run3 > out &
```

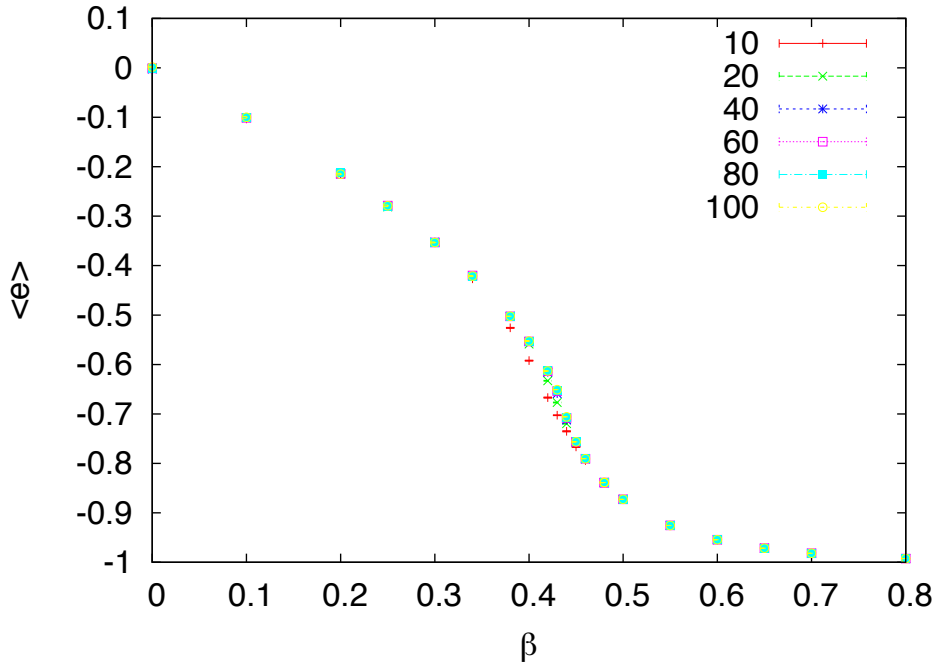



Figure 14.8: The results of the simulations performed by the shell script in the file `run3`. The plot shows the expectation value $\langle e \rangle$.

Then, we can plot the results using `gnuplot`[22]:

```
set xlabel "beta"
set ylabel "<m>"
plot   "<grep '^m 10 '  out" u 3:4:5 with errorbars title " 10"
replot "<grep '^m 20 '  out" u 3:4:5 with errorbars title " 20"
replot "<grep '^m 40 '  out" u 3:4:5 with errorbars title " 40"
replot "<grep '^m 60 '  out" u 3:4:5 with errorbars title " 60"
replot "<grep '^m 80 '  out" u 3:4:5 with errorbars title " 80"
replot "<grep '^m 100 ' out" u 3:4:5 with errorbars title "100"
```

The above commands plot the magnetization.

```
set ylabel "chi"
set log y
plot   "<grep '^m 10 '  out" u 3:6:7 with errorbars title " 10"
replot "<grep '^m 20 '  out" u 3:6:7 with errorbars title " 20"
```

[22]These are `gnuplot` commands, even though we do not follow the usual convention to show the prompt `gnuplot>` explicitly

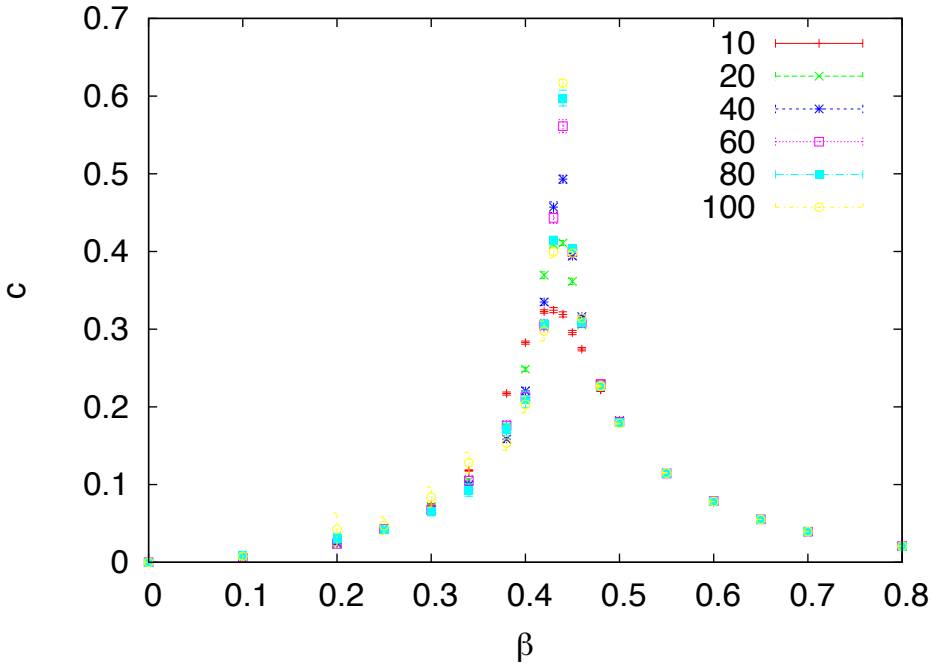



Figure 14.9: The results of the simulations performed by the shell script in the file `run3`. The plot shows the specific heat $c$ which is shown to be almost independent of $L$ away from the critical region, whereas in the critical region it increases according to equation (13.8).

```
replot "<grep '^m 40 ' out" u 3:6:7 with errorbars title " 40"
replot "<grep '^m 60 ' out" u 3:6:7 with errorbars title " 60"
replot "<grep '^m 80 ' out" u 3:6:7 with errorbars title " 80"
replot "<grep '^m 100 ' out" u 3:6:7 with errorbars title "100"
```

The above commands plot the magnetic susceptibility.

```
set ylabel "<e>"
plot   "<grep '^e 10 ' out" u 3:4:5 with errorbars title " 10"
replot "<grep '^e 20 ' out" u 3:4:5 with errorbars title " 20"
replot "<grep '^e 40 ' out" u 3:4:5 with errorbars title " 40"
replot "<grep '^e 60 ' out" u 3:4:5 with errorbars title " 60"
replot "<grep '^e 80 ' out" u 3:4:5 with errorbars title " 80"
replot "<grep '^e 100 ' out" u 3:4:5 with errorbars title "100"
```

The above commands plot the energy.

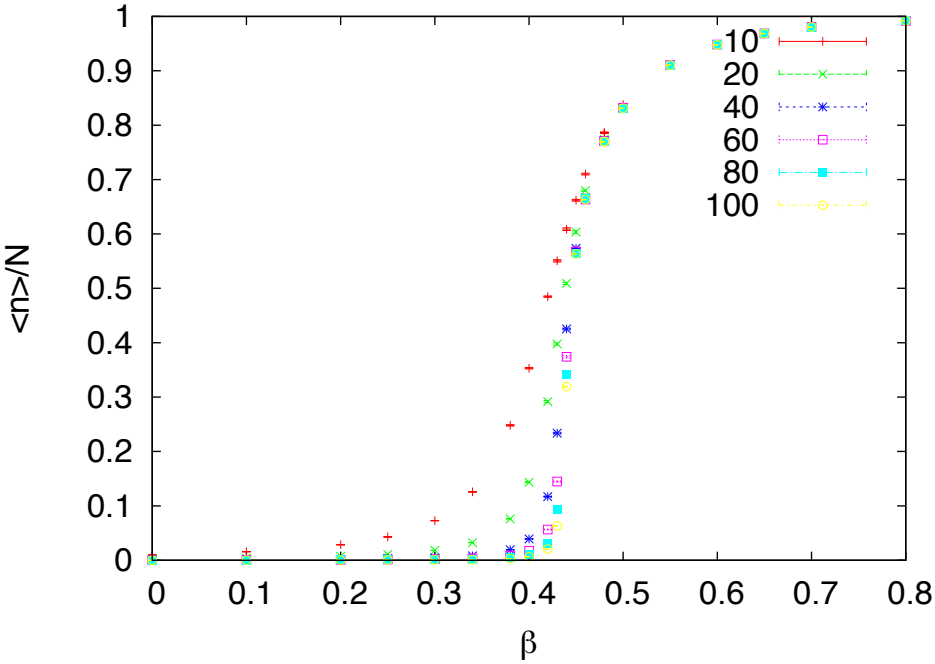



Figure 14.10: The results of the simulations performed by the shell script in the file `run3`. The plot shows $\langle n \rangle / N$.

```
set ylabel "c"
plot   "<grep '^e 10 '  out" u 3:6:7 with errorbars title " 10"
replot "<grep '^e 20 '  out" u 3:6:7 with errorbars title " 20"
replot "<grep '^e 40 '  out" u 3:6:7 with errorbars title " 40"
replot "<grep '^e 60 '  out" u 3:6:7 with errorbars title " 60"
replot "<grep '^e 80 '  out" u 3:6:7 with errorbars title " 80"
replot "<grep '^e 100 ' out" u 3:6:7 with errorbars title "100"
```

The above commands plot the specific heat.

```
set ylabel "<n>/N"
plot   "<grep '^n 10 '  out" u 3:4:5 with errorbars title " 10"
replot "<grep '^n 20 '  out" u 3:4:5 with errorbars title " 20"
replot "<grep '^n 40 '  out" u 3:4:5 with errorbars title " 40"
replot "<grep '^n 60 '  out" u 3:4:5 with errorbars title " 60"
replot "<grep '^n 80 '  out" u 3:4:5 with errorbars title " 80"
replot "<grep '^n 100 ' out" u 3:4:5 with errorbars title "100"
```

The above commands plot $\langle n \rangle / N$.

## 14.6 Autocorrelation Times

In the case of the Metropolis algorithm, the "unit of time" in the Monte Carlo simulation is one "sweep", which is equal to $N$ *attempted* spin flips. In the case of the Wolff algorithm, the size of the clusters is a stochastic variable, which depends on temperature. Therefore, flipping the spins of a cluster is not a convenient unit of time, and we define:

$$(1 \text{ sweep}) = \frac{N}{\langle n \rangle}(\text{Wolff cluster updates}) \tag{14.30}$$

This definition of a sweep can be compared to a Metropolis sweep defined as $N$ *accepted* spin flips[23]. For convenience, we also use the $\beta$–dependent unit of time equal to one Wolff cluster update. We use the notation $\tau_{\mathcal{O}}^{W}$ when the autocorrelation time of $\mathcal{O}$ is measured in Wolf cluster updates, and $\tau_{\mathcal{O}}$ when using the definition (14.30). Their relation is:

$$\tau_{\mathcal{O}} = \tau_{\mathcal{O}}^{W} \frac{\langle n \rangle}{N} \,. \tag{14.31}$$

We simulate the Ising model for $L = 10, 20, 40, 60, 80$ and $100$ at $\beta = 0.4407$ using the Wolff algorithm. We construct $5 \times 10^6$ Wolff clusters. The results are written to files with names `outL${L}b0.4407`. We also perform simulations using the Metropolis algorithm with $10 \times 10^6$ sweeps. The results are written to files with names `outL${L}b0.4407met`. The following shell script makes life easier:

```
#!/bin/tcsh -f
set Ls      = (10 20 40 60 80 100)
set beta    = 0.4407
set nsweeps = 5000000
set start   = "-s 1 -S 3423"
# Wolf cluster algorithm:
foreach  L    ($Ls)
 ./is -w -L $L -b $beta -n $nsweeps $start > outL${L}b${beta}
 # Mean cluster size <n>/N
 grep '#clu' outL${L}b${beta}              | \
 awk -v L=$L 'NR>10000{print $2/(L*L)}'    | \
 ./jack -d $nsweeps | grep -v '#'          | \
 awk -v L=$L -v b=$beta '{print "n",L,b,$1,$2}'
end
```

[23]The two definitions of a sweep in the Metropolis algorithm differ by a factor equal to the average acceptance rate. Both definitions are used in the bibliography, and the reader (as well as the author) of a scientific article should be aware of that.

```
# Metropolis algorithm
set nsweeps = 10000000
foreach  L    ($Ls)
 ./is -L $L -b $beta -n $nsweeps $start   > outL${L}b${beta}met
end
```

We compile the file `autoc.f90` from the `Tools` directory and the executable file is named `autoc` and copied to the current working directory. Then, the following shell script calculates the autocorrelation functions $\rho_m(t)$:

```
#!/bin/tcsh -f
set Ls = (10 20 40 60 80 100)
set b  = 0.4407
# Wolff
set tmax  = 1000
set ndata = 5000000
foreach L ($Ls)
 set f = outL${L}b${b}
 grep -v '#' $f | \
 awk -v L=$L      \
  'BEGIN{N=L*L}NR>100000{print  ($2>0)?($2/N):(-$2/N)}'|\
 ./autoc -t $tmax -n $ndata> $f.rhom
end
# Metropolis
set tmax  = 8000
set ndata = 10000000
foreach L ($Ls)
 set f = outL${L}b${b}met
 grep -v '#' $f | \
 awk -v L=$L      \
  'BEGIN{N=L*L}NR>100000{print  ($2>0)?($2/N):(-$2/N)}'|\
 ./autoc -t $tmax  -n $ndata> $f.rhom
end
```

We throw away 100 000 sweeps for thermalization. The results are written to files whose names have file extension `.rhom`. The function $\rho_m(t)$ is fitted to (13.51) using three autocorrelation times according to the discussion in appendix 13.7. The results are shown in table 14.1[24]

From (14.10) we expect that $\tau_m \sim L^z$ where $z$ is the dynamic exponent. $z$ can be calculated by the `gnuplot` commands:

[24]Notice the difference between the results of the Metropolis algorithm and the ones shown in appendix 13.7. The difference is due to a fivefold increase in the statistics and shows that the real error in the calculation of $\tau$ includes systematic errors that have been neglected.

| $L$ | $\tau_m^W$ | $\langle n\rangle/N$ | $\tau_m$ | $\tau_{m,\text{Metropolis}}$ |
|---|---|---|---|---|
| 10 | 2.18(2) | 0.6124(2) | 1.33(1) | 16.1(1) |
| 20 | 3.48(5) | 0.5159(1) | 1.80(3) | 70.7(4) |
| 40 | 5.10(6) | 0.4342(2) | 2.21(3) | 330(6) |
| 60 | 6.12(6) | 0.3927(2) | 2.40(2) | 795(5) |
| 80 | 7.33(7) | 0.3653(3) | 2.68(3) | 1740(150) |
| 100 | 8.36(6) | 0.3457(1) | 2.89(2) | 2660(170) |

Table 14.1: The autocorrelation times for the magnetization calculated as described in the text. The second column contains the autocorrelation time $\tau_m^W$ for the Wolff algorithm, using one cluster update as the unit of time. The fourth column contains $\tau_m$ in sweeps according to (14.30) and we have that $\tau_m = \tau_m^W \langle n\rangle/N$ (see (14.31)). The fifth column contains the autocorrelation times for the Metropolis algorithm in units of sweeps defined as $N$ attempted spin flips.

```
gnuplot> tau(x) = c*x**z
gnuplot> fit tau(x) "autoc.dat" u 1:2:3 via c,z
gnuplot> plot "autoc.dat" u 1:2:3 w e t 'W steps   ", tau(x)
gnuplot> fit tau(x) "autoc.dat" u 1:6:7 via c,z
gnuplot> plot "autoc.dat" u 1:6:7 w e t 'W sweeps  ", tau(x)
gnuplot> fit tau(x) "autoc.dat" u 1:8:9 via c,z
gnuplot> plot "autoc.dat" u 1:8:9 w e t "Metropolis", tau(x)
```

The exponent $z$ is calculated for the Wolff algorithm in Wolff steps and Wolff sweeps. The results are

$$\tau_m^W \sim L^{z^W}\,, \qquad z^W = 0.54 \pm 0.02 \tag{14.32}$$

$$\tau_m \sim L^{z}\,, \qquad z = 0.29 \pm 0.02 \tag{14.33}$$

$$\tau_{m,\text{Metropolis}} \sim L^{z}\,, \qquad z = 2.21 \pm 0.02 \tag{14.34}$$

The plots are shown in figures 14.11-14.13. The values of $z$ reported in the bibliography are $0.50(1)$, $0.25(1)$ and $2.167(1)$ respectively [4, 57, 64]. We can obtain better results by increasing the statistics and the lattice size and this is left as an exercise for the reader.

We also mention the relation between the dynamic exponents given by equations (14.32) and (14.33). From (14.29) $\chi = \beta\langle n\rangle$, (13.10) $\chi \sim |t|^{-\gamma}$, and (13.6) $\xi \sim |t|^{-\nu}$ and using $\xi \sim L$, valid in the critical region, we obtain

$$\tau_m = \tau_m^W \frac{\langle n\rangle}{L^2} \sim L^{z^W} \frac{L^{\gamma/\nu}}{L^2} = L^{z^W+\gamma/\nu-2}\,, \tag{14.35}$$

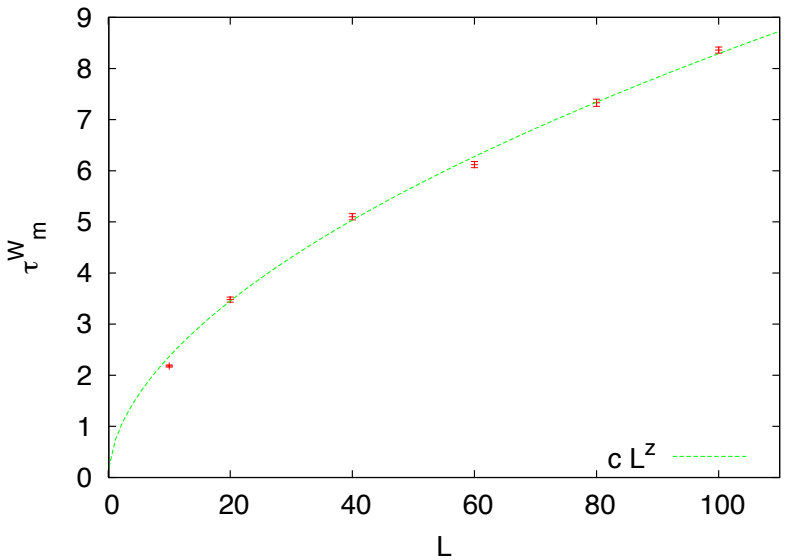



Figure 14.11: Autocorrelation times $\tau_m^W$ for the magnetization using the Wolff algorithm at $\beta = 0.4407$. The unit of time is one Wolff cluster update. The dynamic exponent is calculated from the fit to $cL^{z^W}$ which gives $z^W = 0.54(2)$.

where we assumed that $\tau_m^W \sim L^{z^W}$, $z^W \equiv 0.54(2)$ and $\tau_m \sim L^z$. Therefore

$$z = z^W + \frac{\gamma}{\nu} - 2\,. \tag{14.36}$$

Using the values given in (13.12), $\gamma = 7/4$, $\nu = 1$, we obtain

$$z = z^W - \frac{1}{4}\,, \tag{14.37}$$

which is in agreement, within error, with the calculated values and the values in the bibliography.

## 14.7 Temperature Scaling

In this section we will discuss the extent to which relations (14.3)–(14.5) can be used for the calculation of the critical exponents $\gamma$, $\alpha$ and $\beta$. The result is that, although using them it is possible to compute correct results, these relations are not the best choice[25]. In order to see clear scaling and

[25]Note that for the Ising model on the square lattice, the critical temperature is exactly known. In a model where it is not known we have larger systematic errors than the ones

| $L$ | $\gamma(t<0)$ | $\gamma(t>0)$ |
|---|---|---|
| 40 | 1.7598(44) | 1.730(17) |
| 60 | 1.7455(24) | 1.691(14) |
| 80 | 1.7409(21) | 1.737(12) |
| 100 | 1.7420(24) | 1.7226(75) |
| 120 | 1.7390(15) | 1.7725(69) |
| 140 | 1.7390(23) | 1.7354(72) |
| 160 | 1.7387(10) | 1.746(17) |
| 200 | 1.7380(11) | 1.759(15) |
| 500 | 1.7335(8) | 1.7485(83) |

Table 14.2: Calculation of the critical exponent $\gamma$ from fitting the data shown in figures 14.14 and 14.15. The second column contains the results for $\beta > \beta_c(t<0)$ and the third one for $\beta < \beta_c(t>0)$. The parentheses report the *statistical* errors of the fits and not the systematic. We expect that $\gamma = 7/4$.

| $L$ | $\beta(t<0)$ | $\beta_+(t>0)$ |
|---|---|---|
| 40 | 0.1101(7) | 0.1122(29) |
| 60 | 0.1129(5) | 0.1102(19) |
| 80 | 0.1147(5) | 0.1118(21) |
| 100 | 0.1175(3) | 0.1170(11) |
| 120 | 0.1167(4) | 0.1172(16) |
| 140 | 0.1190(2) | 0.1187(19) |
| 160 | 0.1191(4) | 0.1134(20) |
| 200 | 0.1205(10) | 0.1138(24) |
| 500 | 0.1221(2) | 0.1294(50) |

Table 14.3: The calculation of the critical exponent $\beta$ from fitting the data shown in figures 14.16. The second column contains the results for $\beta > \beta_c(t<0)$ and the third for $\beta < \beta_c(t>0)$. The parentheses report the *statistical* errors of the fits and not the systematic. We expect that $\beta = \beta_+ = 1/8$.

| $L$ | $\gamma/\nu$ | $\beta/\nu$ |
|---|---|---|
| 40–100 | 1.754(1) | 0.1253(1) |
| 140–1000 | 1.740(2) | 0.1239(3) |
| 40–1000 | 1.749(1) | 0.1246(1) |

Table 14.4: The critical exponents $\gamma/\nu$ and $\beta/\nu$ given by the finite size scaling relations (14.7) and (14.9). The first column contains the range in $L$ included in the fits of $\chi(\beta_c, L)$ and $\langle m\rangle(\beta_c, L)$ to $aL^g$.

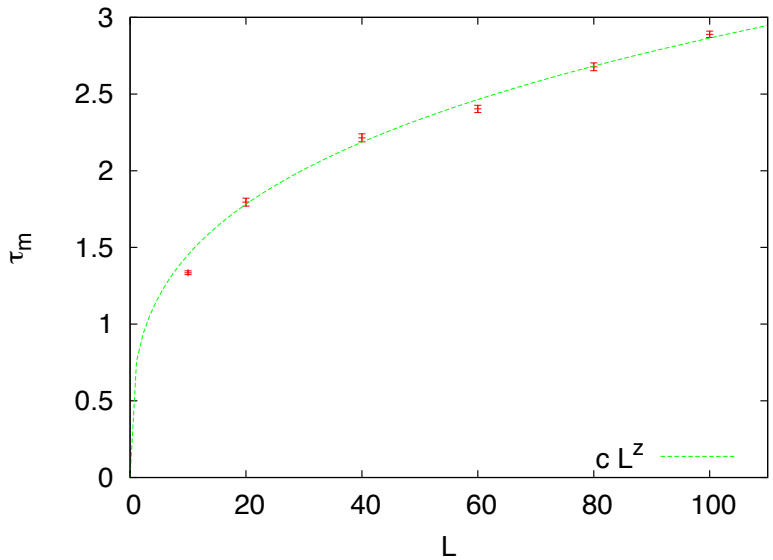



Figure 14.12: Autocorrelation times $\tau_m$ for the magnetization using the Wolff algorithm at $\beta = 0.4407$. The unit of time is one Wolff sweep. The dynamic exponent is calculated from the fit to $cL^z$, which gives $z = 0.29(2)$.

reduce finite size effects, we have to consider $t \ll 1$ and large $L$. The results depend strongly on the choice of range of the data included in the fits. The systematic errors are large and the results in some cases plain wrong[26].

We simulate the Ising model for $L = 40$, $60$, $80$, $100$, $120$, $140$, $160$, $200$ and $500$. The temperatures chosen correspond to small enough $t$ in order to observe scaling. For the values of $\beta$ used in the simulations, see the shell scripts in the accompanying software.

First we compute the exponent $\gamma$ from the relation (14.3). For given $L$, we fit the data for $\chi(t)$ for an appropriate range of $|t|$ to the function $a\,|t|^{-\gamma}$, which has two fitting parameters, $\gamma$ and $a$. We determine the range of $t$ where $\chi(t)$ gives a linear plot in a log–log scale[27]. For large $|t|$, we observe deviations from the linear behavior and for very small $|t|$ we

discussed here. The numerical calculation of the critical temperature we will discussed in a following section.

[26]In [4] it is mentioned that the random field Ising model exhibits pseudoscaling for a range of $t$ and for even smaller $t$ there is a crossover to a different scaling that gives the correct critical exponent. See also [65], [66].

[27]The fit can also be done by linearly fitting the points $(\log|t_i|, \log\chi(t_i))$ to a straight line.

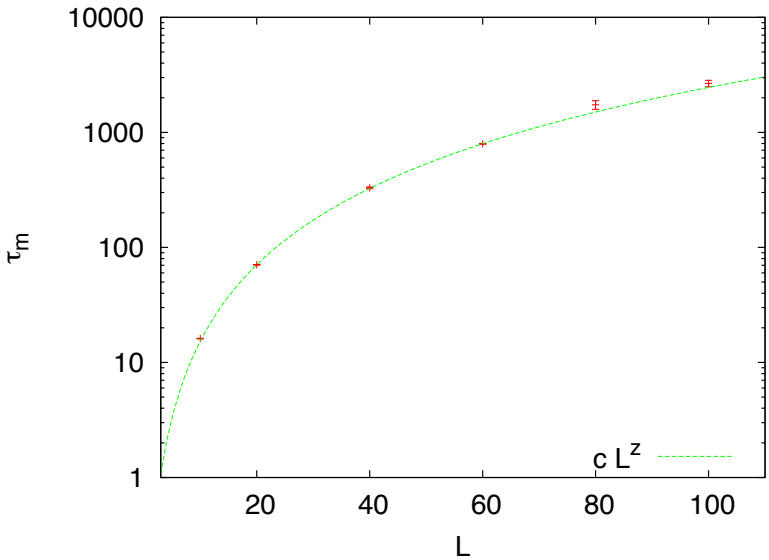



Figure 14.13: Autocorrelation times $\tau_{m,\text{Metropolis}}$ for the magnetization using the Metropolis algorithm at $\beta = 0.4407$. The unit of time is a Metropolis sweep defined by $N$ attempted spin flips. The dynamic exponent is calculated from the fit to $cL^z$, which gives $z = 2.21(2)$.

observe finite size effects when $\xi \approx L$. As $L$ increases, finite size effects decrease, and the data get closer to the asymptotic behavior $|t|^{-\gamma}$ for even smaller $|t|$. The results are more clear for $\beta > \beta_c(t < 0)$, because for $t > 0$ the fluctuations near the pseudocritical temperature $\beta_c(L) < \beta_c$ are larger and the finite size effects are larger.

Table 14.2 shows the results for the exponent $\gamma$ for all the measured values of $L$. The errors reported are the statistical errors of the fits, which are smaller than the systematic errors coming from the choice or range of $t$ of the data included in the fits. One has to vary this range as long as the $\chi^2$/dof of the fit remains acceptable, and the resulting variation in the values of the parameters has to be included in the estimate of the error. Sometimes, this method gives an overestimated error, and it is a matter or experience to decide which parameter values to include in the estimate. For example, figures 14.14 and 14.15 show that the acceptable range of fitting becomes more clear by studying $\chi(t)$ for increasing $L$. As $L$ increases, the points approach the asymptotic curve even closer. Even though for fixed $L$ one obtains acceptable power fits over a larger range of $t$, by studying the large $L$ convergence, we can determine the scaling region with higher accuracy. Another point to consider is whether the

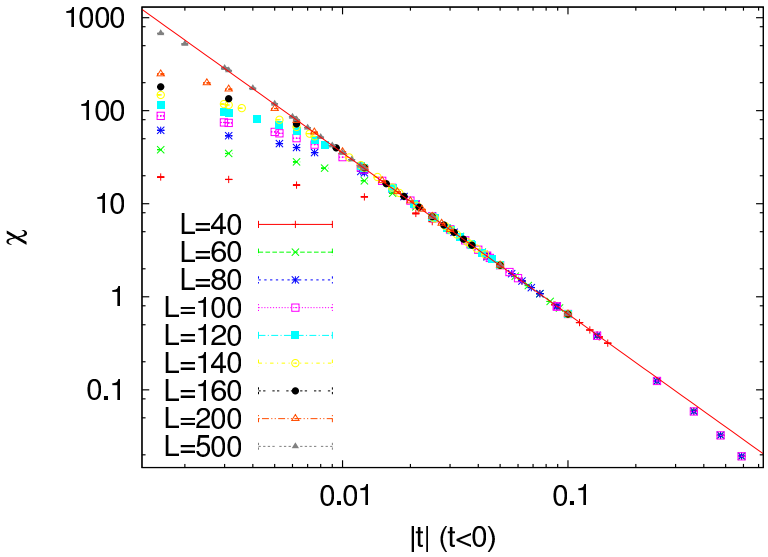



Figure 14.14: The magnetic susceptibility $\chi(t, L)$ in the scaling region according to equation (14.3). The straight line is the fit to this relation for the largest lattice. We observe that finite size effects decrease as $L$ increases and that the range of temperatures included in the fit extends to smaller $|t|$. The data is for $\beta > \beta_c(t < 0)$ and the critical point is approached from the ordered phase.

parameters of the fits have reasonable values. For example, even though the value of $a$ is unknown, it is reasonable to expect that its value is of order $\sim 1$. By taking all these remarks into consideration we obtain

$$\gamma = 1.74 \pm 0.02 \qquad (t < 0)\,, \tag{14.38}$$

$$\gamma = 1.73 \pm 0.04 \qquad (t > 0)\,. \tag{14.39}$$

Next, we compute the critical exponent $\beta$ using relation (14.5). This relation is valid as we approach the critical point from the low temperature phase, $\beta > \beta_c$ or $t < 0$. In the thermodynamic limit, the magnetization is everywhere zero for all $\beta < \beta_c$. For a finite lattice $\langle m \rangle > 0$, and it is reasonable to expect a scaling of the form

$$\langle m \rangle \sim |t|^{\beta_+ - 1}\,, \qquad t > 0\,, \tag{14.40}$$

where $\beta_+$ is defined so that

$$\beta_+ = \beta = 1/8\,. \tag{14.41}$$

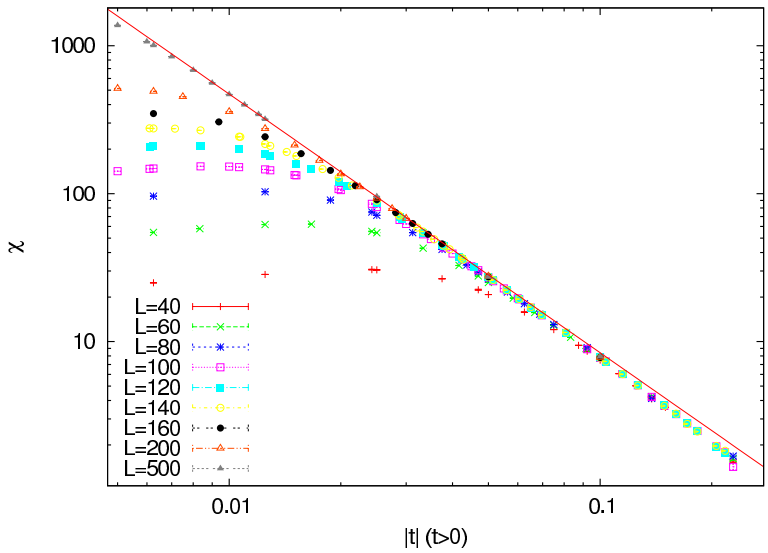



Figure 14.15: The magnetic susceptibility $\chi(t, L)$ in the scaling region according to equation (14.3). The straight line is the fit to this relation for the largest lattice. We observe that finite size effects decrease as $L$ increases and that the range of temperatures included in the fit extends to smaller $|t|$. The data is for $\beta < \beta_c(t > 0)$ and the critical point is approached from the disordered phase. Finite size effects are larger for $t < 0$ due to the larger fluctuations at the pseudocritical point $\beta_c(L) < \beta_c$.

By following the same procedure, we calculate the exponents $\beta$ and $\beta_+$ shown in table 14.3. By taking the systematic errors described above into consideration, we find that

$$\beta = 0.121 \pm 0.003 \qquad t < 0\,, \tag{14.42}$$

$$\beta_+ = 0.120 \pm 0.007 \qquad t < 0\,, \tag{14.43}$$

which should be compared to the expected values $\beta = \beta_+ = 1/8$.

The calculation of the exponent $\alpha$ needs special care. The expected value is $\alpha = 0$. This does not imply that $c \sim$ const. but that [28]

$$c \sim |\log|t|| \ . \tag{14.44}$$

In this case, we find that the data is better fitted to the above relation instead of being fitted to a power. This can be seen pictorially by making

[28]This does not exclude more exotic behaviors of logarithmic powers or logarithms of logarithms etc. This needs to be studied carefully when the analytic result is not known.

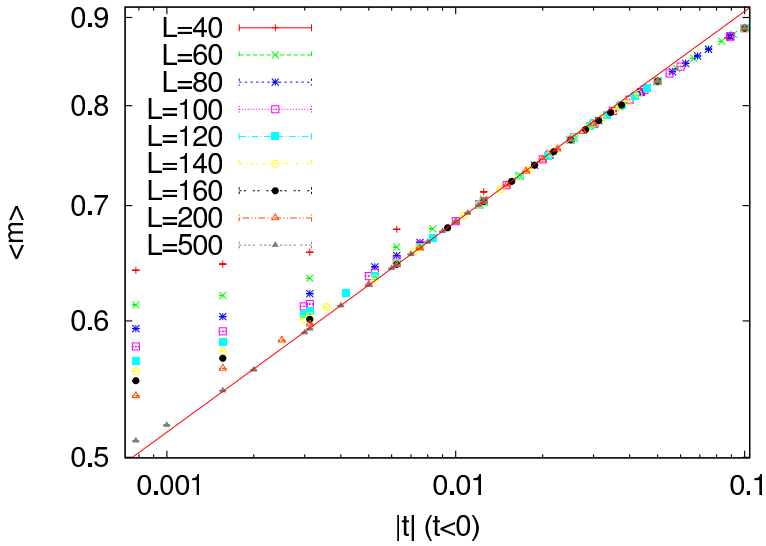



Figure 14.16: The magnetization $\langle m\rangle(t, L)$ in the scaling region according to equation (14.5). The straight line is the fit to this relation for the largest lattice. We observe that finite size effects decrease as $L$ increases and that the range of temperatures included in the fit extends to smaller $|t|$. The data is for $\beta > \beta_c(t < 0)$ and the critical point is approached from the ordered phase.

a log–log plot and comparing it to a $c - |\log|t||$ plot. We see that the second choice leads to a better linear plot than the first one. A careful study will compute the quality of the fits and choose the better model this way. This is left as an exercise for the reader.

## 14.8 Finite Size Scaling

In this section we will calculate the critical exponents by using relations (14.7)-(14.9), i.e. by using the asymptotic scaling of $\chi(\beta = \beta_c, L)$, $c(\beta = \beta_c, L)$ and $\langle m\rangle(\beta = \beta_c, L)$ with increasing system size $L$. This is called "finite size scaling".

In order to calculate the exponent $\gamma/\nu$ given by equation (14.7), we calculate the magnetic susceptibility at the known $\beta_c$ for increasing values of $L$. We fit the results $\chi(\beta_c, L)$ to the function $aL^g$ and calculate the fitting parameters $a$ and $g$. Then, we compare the computed value of $g$ to the expected value of $\gamma/\nu = 7/4 = 1.75$. In this procedure we have to decide which values of $L$ should be included in the fits. The most obvious criterion is to obtain reasonable $\chi^2/\text{dof} \lesssim 1$ and that the error in $g$ and $a$ be small. This is not enough: Table 14.4 shows small variations

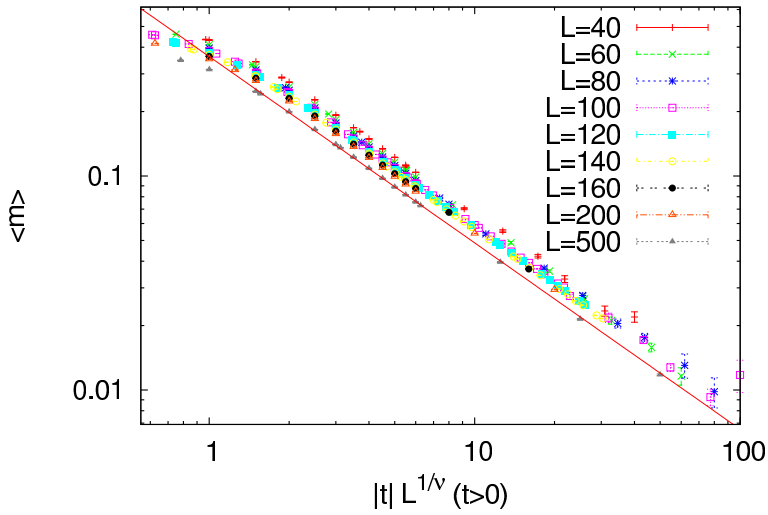



Figure 14.17: The magnetization $\langle m\rangle(t, L)$ in the scaling region fitted to equation (14.40). The straight line is the fit to this relation for the largest lattice. We observe that finite size effects decrease as $L$ increases and that the range of temperatures included in the fit extends to smaller $|t|$. The data is for $\beta < \beta_c(t > 0)$ and the critical point is approached from the disordered phase.

in the obtained values of $\gamma/\nu$, if we consider different fit ranges. These variations give an estimate of the systematic error which enters in the calculation. Problem 9 is about trying this calculation yourselves. Table 14.4 shows the results, and figure 14.20 shows the corresponding plot. The final result, which includes also an estimate of the systematic errors, is

$$\frac{\gamma}{\nu} = 1.748 \pm 0.005\,. \tag{14.45}$$

For the calculation of the exponent $\beta/\nu$ given by equation (14.9), we compute the magnetization $\langle m\rangle(\beta_c, L)$ at the critical temperature and repeat the same analysis. The result is

$$\frac{\beta}{\nu} = 0.1245 \pm 0.0006\,. \tag{14.46}$$

Equation (14.9) gives the exponent $\alpha/\nu$. But the expected value $\alpha = 0$ leads, in analogy with equation (14.44), to

$$c(\beta_c, L) \sim \log L\,. \tag{14.47}$$

This relation is shown in figure 14.22. The vertical axis is *not* in a logarithmic scale whereas the horizontal is. The linear plot of the data

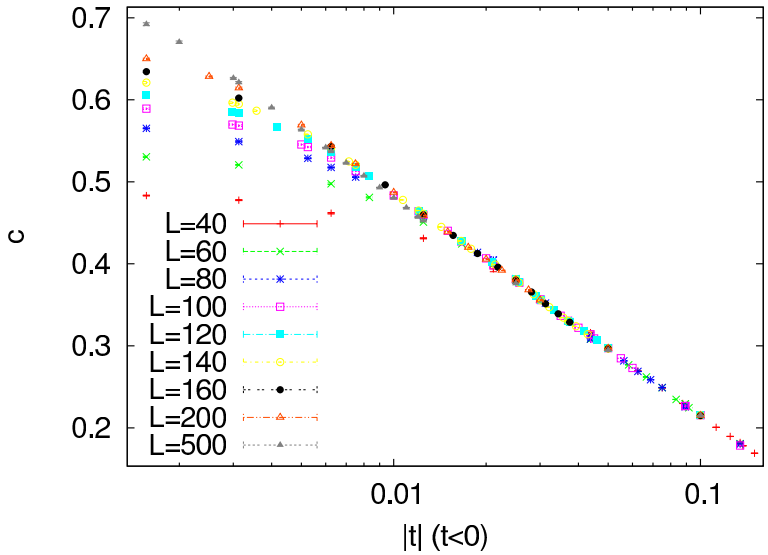



Figure 14.18: The specific heat $c(t, L)$ in the scaling region fitted to equation (14.44). Only the $|t|$ axis is in logarithmic scale. The data is for $\beta > \beta_c(t < 0)$ and the critical point is approached from the ordered phase.

shows consistency with equation (14.47). Problem 9 asks you to show whether the logarithmic fit is better than a fit to a function of the form $cL^a+b$ and appreciate the difficulties that arise in this study. By increasing the statistics, and by measuring for larger $L$, the data in table 14.8 will improve and lead to even clearer conclusions.

We observe that, by using finite size scaling, we can compute the critical exponents more effectively, than by using temperature scaling as in section 14.7. The data follow the scaling relations (14.7)–(14.9) suffering smaller finite size effects[29].

## 14.9 Calculation of $\beta_c$

In the previous sections we discussed scaling in $t$ and $L$ in the critical region. In the calculations we used the exact value of the critical temperature $\beta_c = \log(1 + \sqrt{2})/2$. When $\beta_c$ is not known, the analysis becomes harder and its computation contributes to the total error in the value of the critical exponents. When doing finite size scaling using the scaling

[29]Remember that the instability of the results with respect to the choice of the fitting range is large for the temperature scaling method. When the exact value of the critical temperature is not known, the superiority of finite size scaling is even higher.

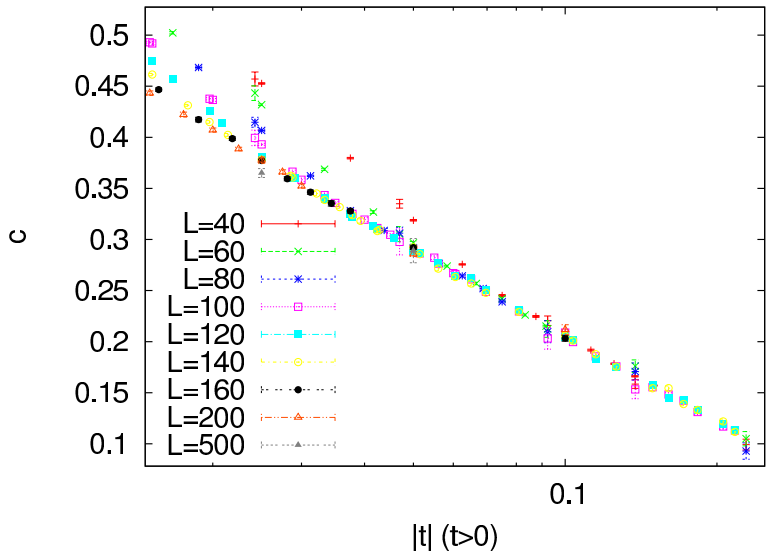



Figure 14.19: The specific heat $c(t, L)$ in the scaling region fitted to equation (14.44). Only the $|t|$ axis is in logarithmic scale. The data is for $\beta < \beta_c(t > 0)$ and the critical point is approached from the disordered phase. The exponent $\nu$ is set equal to 1.

relations (14.7)–(14.9), one has to choose the values of the temperature at which to calculate the left hand sides. So far, these values were computed *at* $\beta_c$. What should we do when $\beta_c$ is not a priori known? A good choice is to use the pseudocritical temperature $\beta_c(L)$, the temperature where the fluctuations of the order parameter $\chi(\beta)$ are at their maximum. Otherwise, we can compute $\beta_c$ according to the discussion in this section and use the computed $\beta_c$ in the finite size scaling analysis.

Both choices yield the same results in the large $L$ limit, even though the finite size effects are different. In fact any value of $\beta$ in the critical region can be used for this calculation. The reason is that as we approach the critical region for given $L$, the correlation length becomes $\xi \sim L$ and finite size effects become important. This is the behavior that characterizes the *pseudocritical region* of the finite $L$ system. The pseudocritical region becomes narrower as $L$ becomes larger. Any value of $\beta$ in this region will give us observables that scale at large $L$, but the best choice is

$$\chi(\beta_c(L), L) \equiv \chi_{\max}(L)\,. \tag{14.48}$$

In this case, the values on the left hand sides of (14.7)–(14.9) should be taken at $\beta = \beta_c(L)$.

The definition of $\beta_c(L)$ in not unique. One could use, for example,

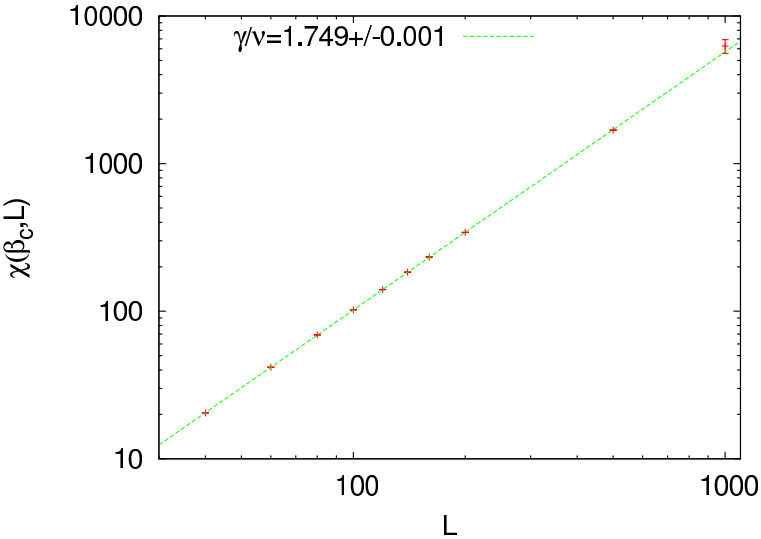



Figure 14.20: The magnetic susceptibility $\chi(\beta_c, L)$ at the critical temperature for different values of $L$. The axes are in a logarithmic scale and the linear plots are consistent with the power fit $\chi(\beta_c, L) = cL^g$. The value of $g$ computed by the fits is consistent with the critical exponent $\gamma/\nu$ given by equation (14.7).

the maximum of the specific heat

$$c(\beta_c'(L), L) \equiv c_{\max}(L)\,, \tag{14.49}$$

which defines a different $\beta_c'(L)$. Of course $\lim_{L\to\infty} \beta_c(L) = \lim_{L\to\infty} \beta_c'(L) = \beta_c$ and both choices will yield the same results for large $L$. The speed of convergence and the errors involved in the calculation of the pseudocritical temperatures are different in each case and there is a preferred choice, which in our case is $\beta_c(L)$.

First we calculate $\beta_c$. When we are in the pseudocritical region we have that $\xi \approx L$, therefore (14.2) gives

$$|t| = \left|\frac{\beta_c - \beta_c(L)}{\beta_c}\right| \sim \xi^{-\frac{1}{\nu}} \sim L^{-\frac{1}{\nu}} \Rightarrow \beta_c(L) = \beta_c - \frac{c}{L^{\frac{1}{\nu}}}\,. \tag{14.50}$$

The calculation is straightforward to do: First we measure the magnetic susceptibility. For each $L$ we determine the pseudocritical region and we calculate $\beta_c(L)$ and the corresponding maximum value $\chi_{\max}$. In order to do that, we should take many measurements around $\beta_c(L)$. We have to be very careful in determining the autocorrelation time (which increases as $\tau \sim L^z$), so that we can control the number of independent measurements and the thermalization of the system. We use the relation (14.50) and fit

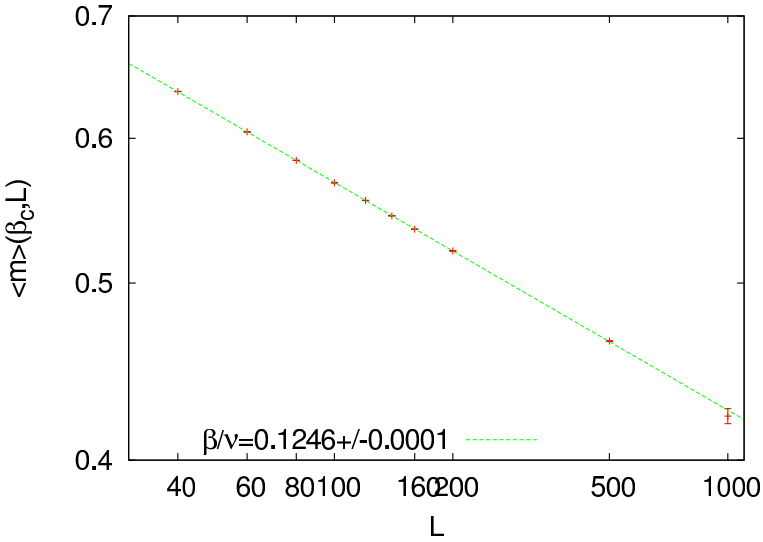



Figure 14.21: The magnetization $\langle m\rangle(\beta_c, L)$ at the critical temperature for different values of $L$. The axes are in a logarithmic scale and the linear plots are consistent with the power fit $\langle m\rangle(\beta_c, L) = cL^g$. The value of $g$ computed by the fits is consistent with the critical exponent $\beta/\nu$ given by equation (14.9).

the results to $a - c/L^b$, and from the calculated parameters $a, b$ and $c$ we compute $\beta_c = a$, $\nu = 1/b$. In cases where one of the parameters $\beta_c$, $\nu$ is known independently, then it is kept constant during the fit.

The results are shown in figure 14.23 where we plot the numbers contained in table 14.5. The final result is:

$$\begin{aligned} \beta_c &= 0.44066 \pm 0.00003 \\ \frac{1}{\nu} &= 1.006 \pm 0.017\,. \end{aligned}$$

This can be compared to the known values $\beta_c = \log(1+\sqrt{2})/2 \approx 0.44069$ and $1/\nu = 1$.

This process is repeated for the pseudocritical temperatures $\beta_c'(L)$ and the maximum values of the specific heat $c_{\max}$. The results are shown in figure 14.24. The final result is:

$$\begin{aligned} \beta_c &= 0.44062 \pm 0.00008 \\ \frac{1}{\nu} &= 1.09 \pm 0.18\,. \end{aligned}$$

Figure 14.24 and the results reported above show that the calculation using the specific heat gives results compatible with (14.51), but that they

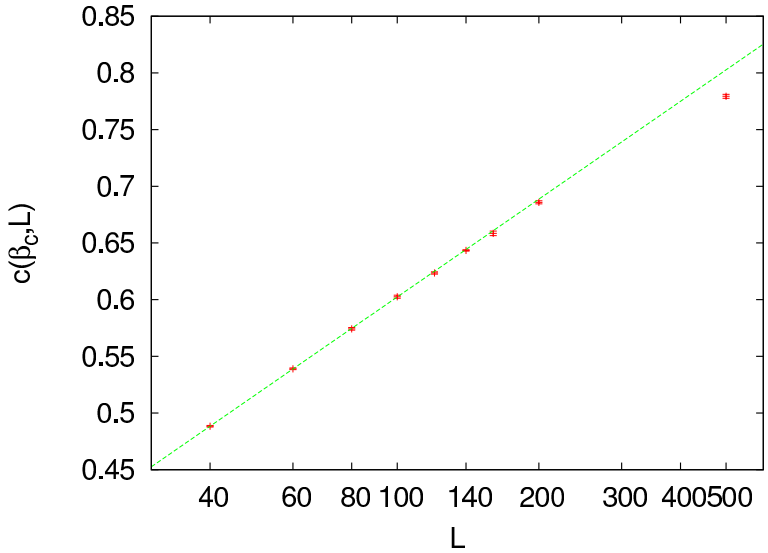



Figure 14.22: The specific heat $c(\beta_c, L)$ at the critical temperature for different values of $L$. The horizontal axis is in a logarithmic scale and the linear plot is consistent with the scaling relation $c(\beta_c, L) = c \log L$. The result is consistent with the expectation $\alpha = 0$ (see equation (14.8)).

are less accurate. The values of the specific heat around its maximum are more spread and more noisy than the ones of the magnetic susceptibility.

From the maxima of the magnetic susceptibility $\chi_{\max}(L)$ we can calculate the exponent $\gamma/\nu$. Their values are shown in table 14.5. The data are fitted to $aL^b$, according to the asymptotic relation (14.9), with $a$ and $b$ being fitting parameters. We find very good scaling, therefore our data are in the asymptotic region. The result is

$$\frac{\gamma}{\nu} = 1.749 \pm 0.001\,, \tag{14.51}$$

which is consistent with the analytically computed value $7/4$.

From the maxima of the specific heat we can calculate the exponent $\alpha/\nu$. Since $\alpha = 0$, the form of the asymptotic behavior is given by (14.47). We find that our results are not very well fitted to the function $a \log L$ and it is possible that the discrepancy is due to finite size effects. We add terms that are subleading in $L$ and find that the fit to the function $a \log L + b - c/L$ is very good[30]. If we attempt to fit the data to the function

[30]Our ansatz is justified by the analytic calculations in [67], which compute the corrections to the $\log L$ behavior. These corrections are shown to be given by integer powers of $1/L$: $c = a \log L + \sum_{k=0}^{\infty} c_k/L^k$.

| $L$ | $\beta_c(L)$ | $\chi_{\max}$ | $\beta'_c(L)$ | $c_{\max}$ |
|---|---|---|---|---|
| 40 | 0.4308(4) | 30.68(4) | 0.437(1) | 0.5000(20) |
| 60 | 0.4342(2) | 62.5(1) | 0.4382(7) | 0.5515(15) |
| 80 | 0.4357(2) | 103.5(1) | 0.4388(5) | 0.5865(12) |
| 100 | 0.4368(1) | 153.3(2) | 0.4396(2) | 0.6154(18) |
| 120 | 0.4375(1) | 210.9(2) | 0.4396(4) | 0.6373(20) |
| 140 | 0.43793(13) | 276.2(4) | 0.4397(5) | 0.6554(18) |
| 160 | 0.4382(1) | 349.0(5) | 0.4398(4) | 0.6718(25) |
| 200 | 0.43870(7) | 516.3(7) | 0.4399(2) | 0.6974(17) |
| 500 | 0.43988(4) | 2558(5) | 0.44038(8) | 0.7953(25) |
| 1000 | 0.44028(4) | 8544(10) | 0.44054(8) | 0.8542(36) |

Table 14.5: The pseudocritical temperatures $\beta_c(L)$ and $\beta'_c(L)$ calculated from the maxima of the magnetic susceptibilities $\chi_{\max}$ and the specific heat $c_{\max}$ respectively. The values of the maxima are also shown.

$aL^d+b-c/L$, the quality of the fit is poor and the result for $d$ is consistent with zero. The results are shown in figure 14.26.

## 14.10 Studying Scaling with Collapse

The scaling relations (14.3)–(14.9) are due to the appearance of a unique, dynamical length scale, the correlation length[31] $\xi$. As we approach the critical point, $\xi$ diverges as $\xi \sim |t|^{-\nu}$, and we obtain universal behavior for all systems in the same universality class. If we consider the magnetic susceptibility $\chi(\beta, L)$, its values depend both on the temperature $\beta$, the size of the system $L$ and of course on the details of the system's degrees of freedom and their dynamics. Universality leads to the *assumption* that the magnetic susceptibility of the infinite system in the critical region depends *only* on the correlation length $\xi$. For the finite system in the pseudocritical region, finite size effects suppress the fluctuations when $\xi \sim L$. The length scales that determine the dominant scaling behavior $\chi \sim \xi^{\gamma/\nu}$ are $\xi$ and $L$, therefore the *dimensionless* variable $L/\xi$ is the only independent variable in the scaling functions. In order to obtain the scaling relation $\chi \sim \xi^{\gamma/\nu}$, valid for the infinite system, we only need to

[31]Careful: $\xi = \xi(t)$ is the correlation length of the *infinite* system at temperature $t$ and not the correlation length at finite $L$.

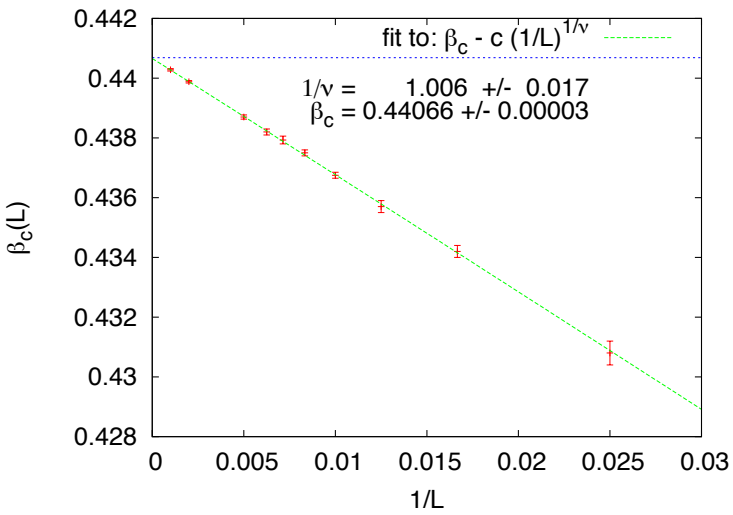



Figure 14.23: Calculation of the critical temperature $\beta_c$ and the critical exponent $\nu$ using relation (14.50). By using the pseudocritical temperatures $\beta_c(L)$ of table 14.5, we fit the data to $a-c(1/L)^b$. From the calculated values of $a$, $b$ and $c$ we calculate $\beta_c = a$ and $1/\nu = b$. The horizontal line is the exact, known value $\beta_c = \log(1+\sqrt{2})/2 = 0.44069\ldots$.

assume that for the finite system[32]

$$\chi = \chi(\beta, L) = \xi^{\gamma/\nu} F_\chi^{(0)}(L/\xi)\,, \tag{14.52}$$

where $F_\chi^{(0)}(z)$ is a function of *one* variable, such that

$$F_\chi^{(0)}(z) = \text{const.} \qquad z \gg 1\,, \tag{14.53}$$

and

$$F_\chi^{(0)}(z) \sim z^{\gamma/\nu} \qquad z \to 0\,. \tag{14.54}$$

Indeed, when $1 \ll \xi \ll L$ ($z \gg 1$) the magnetic susceptibility takes values very close to those of the infinite system, and (14.53) gives $\chi \sim \xi^{\gamma/\nu}$. As $\xi \sim L$, finite size effects enter and (14.54) gives $\chi \sim \xi^{\gamma/\nu}(L/\xi)^{\gamma/\nu} = L^{\gamma/\nu}$. The latter is nothing but (14.7) for the maxima of the magnetic susceptibility of the finite system that we studied in figure 14.25. Therefore the function $F_\chi^{(0)}(z)$ describes how the magnetic susceptibility deviates from scaling due to finite size effects.

The function $F_\chi^{(0)}(z)$ can be calculated using the measurements coming from the Monte Carlo simulation. Since the correlation length is not

[32]For more details see appendix 14.12. The $\beta$ dependence in $\chi(\beta, L)$ enters through the dependence $\xi(\beta)$ of the correlation length of the infinite system in $\beta$.

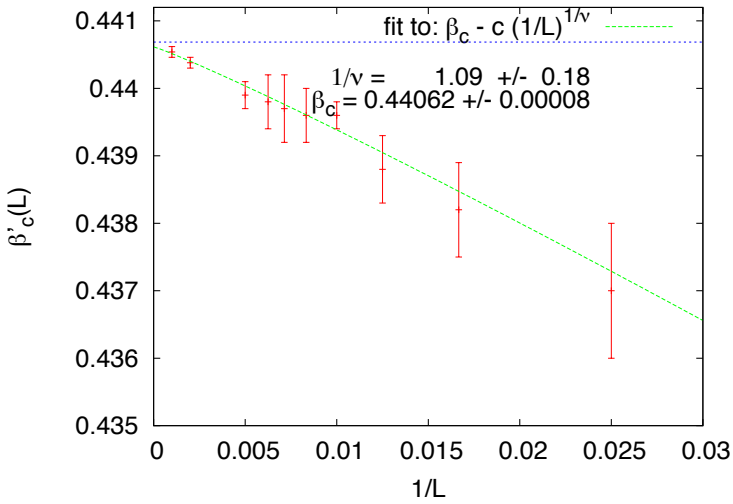



Figure 14.24: Calculation of the critical temperature $\beta_c$ and the critical exponent $\nu$ using relation (14.50). By using the pseudocritical temperatures $\beta_c'(L)$ of table 14.5, we fit the data to $a - c(1/L)^b$. From the calculated values of $a$, $b$ and $c$ we calculate $\beta_c = a$ and $1/\nu = b$. The horizontal line is the exact, known value $\beta_c = \log(1+\sqrt{2})/2 = 0.44069\ldots$.

directly calculated, but appears indirectly in the measurements, we define the dimensionless variable

$$x = L^{1/\nu} t\,, \tag{14.55}$$

where $|x| \sim (L/\xi)^{1/\nu}$ since[33] $\xi \sim |t|^{-\nu}$. We define $F_\chi(x) \propto x^{-\gamma} F_\chi^{(0)}(x^\nu)$ so that (14.52) becomes

$$\chi = L^{\gamma/\nu} F_\chi(x) = L^{\gamma/\nu} F_\chi(L^{1/\nu} t)\,. \tag{14.56}$$

The asymptotic properties of the scaling function $F_\chi(x)$ are determined by the relations (14.53) and (14.54). When $x = L^{1/\nu} t \gg 1$, equation (14.53) is valid for $F_\chi^{(0)}(x^\nu)$ and we obtain $F_\chi^{(0)}(x^\nu) = \text{const}$. From the definition $F_\chi(x) = x^{-\gamma} F_\chi^{(0)}(x^\nu)$ we obtain $F_\chi(x) \sim x^{-\gamma} = (L/\xi)^{-\gamma/\nu}$ and we confirm the scaling property of the magnetic susceptibility in the thermodynamic limit $\chi \sim L^{\gamma/\nu} F_\chi(x) \sim L^{\gamma/\nu}(L/\xi)^{-\gamma/\nu} = \xi^{\gamma/\nu}$. Therefore

$$F_\chi(x) \sim x^{-\gamma} \qquad x \gg 1\,. \tag{14.57}$$

When $x \to 0$, (14.54) is valid and we have that $F_\chi^{(0)}(x^\nu) \sim (x^\nu)^{\gamma/\nu} = x^\gamma$. Then we obtain $F_\chi(x) \propto x^{-\gamma} F_\chi^{(0)}(x^\nu) \sim x^{-\gamma} x^\gamma = \text{const}$. Therefore, we

[33]The absolute value is dropped in the definition of $x$ so that we have a convenient notation for temperatures above and below the critical temperature.

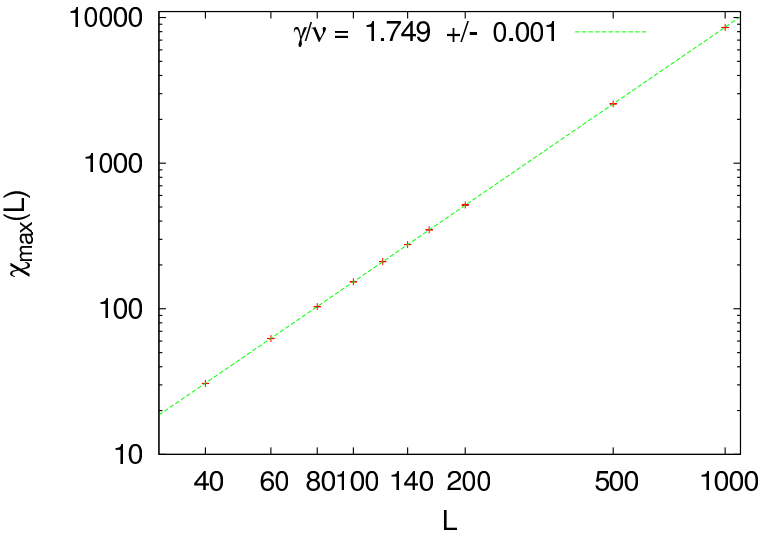



Figure 14.25: Calculation of the critical exponent $\gamma/\nu$ from the maxima of the magnetic susceptibility using the asymptotic scaling (14.9). The values $\chi_{\max}(L)$ are taken from table (14.5) and are fitted to a function of the form $aL^b$. The result of the fit is $\gamma/\nu = 1.749(1)$.

confirm that, when finite size effects are dominant ($x \to 0$), we have that $\chi = L^{\gamma/\nu} F_\chi(x) \sim L^{\gamma/\nu}$. Therefore

$$F_\chi(x) \sim \text{const.} \qquad |x| \ll 1\,. \tag{14.58}$$

By inverting equation (14.56), we can calculate the scaling function from the measurements of the magnetic susceptibility

$$F_\chi(L^{1/\nu} t) = L^{-\gamma/\nu} \chi(\beta, L)\,, \tag{14.59}$$

where $\chi(\beta, L)$ are measurements for temperatures in the pseudocritical region for several values of $L$. When equation (14.59) is valid, then all the measurements fall onto the same curve $F_\chi(x)$ independently of the size $L$! Of course deviations due to finite size effects are expected, especially when $L$ is small. But, as we will see, convergence is quite fast.

Using the above procedure, we can determine the critical temperature $\beta_c$, the exponent $\nu$ and the ratio $\gamma/\nu$ simultaneously! In order to check (14.59), we have to compute the variable $x = L^{1/\nu} t$, for which it is necessary to know $\beta_c$ ($t = (\beta_c - \beta)/\beta_c$) and the exponent $\nu$. For the calculation of $F_\chi$ it is necessary to know $\gamma/\nu$ that appears on the right hand side of (14.59). Relation (14.59) depends quite sensitively on the

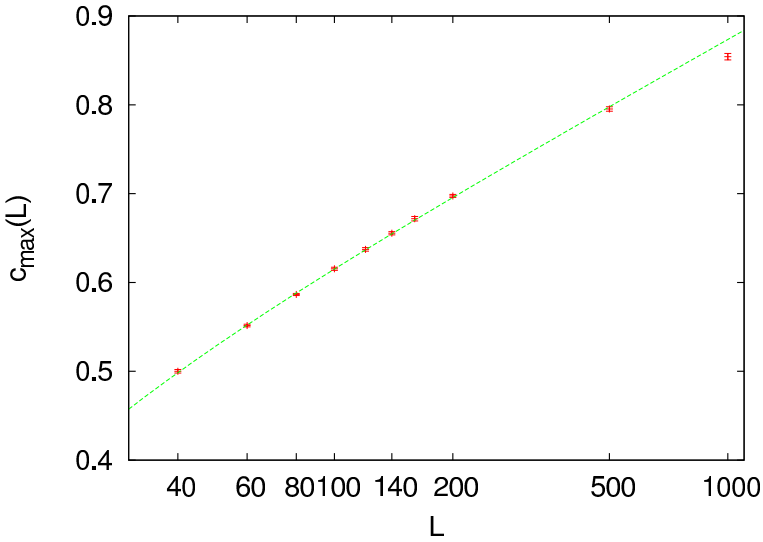



Figure 14.26: Calculation of the critical exponent $\alpha/\nu$ from the maxima of the specific heat using the asymptotic scaling (14.8). The values $c_{\max}(L)$ are taken from table (14.5) and are fitted to a function of the form $a \log L + b - c/L$. We obtain $a = 0.107(3)$, $b = 0.13(1)$ and $c = 1.2(3)$ with $\chi^2/\text{dof} = 0.9$ by fitting for $L = 40, \ldots, 500$. The fit to $aL^d + b - c/L$ gives $d = 0.004(97)$, i.e. an exponent consistent with 0 and somehow weird values for the parameters $a$ and $b$. We conclude that the data is consistent with $\alpha/\nu = 0$.

parameters $\beta_c$, $\nu$ and $\gamma/\nu$ and this way we obtain an accurate method for their calculation.

In order to do the calculation, we set initial values for the parameters $(\beta_c, \nu, \gamma/\nu)$. Using $L$, $\beta$, $\beta_c$ and $\nu$, we calculate the scaling variable $x = L^{1/\nu}t = L^{1/\nu}(\beta_c - \beta)/\beta_c$. Using $\chi(\beta, L)$ and $\gamma/\nu$, we calculate $F_\chi = \chi(\beta, L)/L^{\gamma/\nu}$ and plot the points $(x_i, F_\chi(x_i))$ near the critical region $t \approx 0$. Then we vary $(\beta_c, \nu, \gamma/\nu)$ until the curves for different $L$ collapse onto each other. The optimal collapse determines $(\beta_c, \nu, \gamma/\nu)$.

The collapse of the curves that are constructed from the points $(L_i^{1/\nu}(\beta_c - \beta_i)/\beta_c, L_i^{-\gamma/\nu}\chi(\beta_i, L_i))$ for different $L$ is the most efficient method for studying scaling in the critical region. Figure 14.27 shows the function $F_\chi(x)$ for the known values of the parameters $(\beta_c, \nu, \gamma/\nu) = (\ln(1 + \sqrt{2})/2, 1, 7/4)$. Small variations of the parameters lead to a sharp change of the quality of the collapse. We can make a quick and dirty estimate of the accuracy of the method by varying *one* of the parameters, and look for a visible deviation from all data collapsing onto a single curve. The

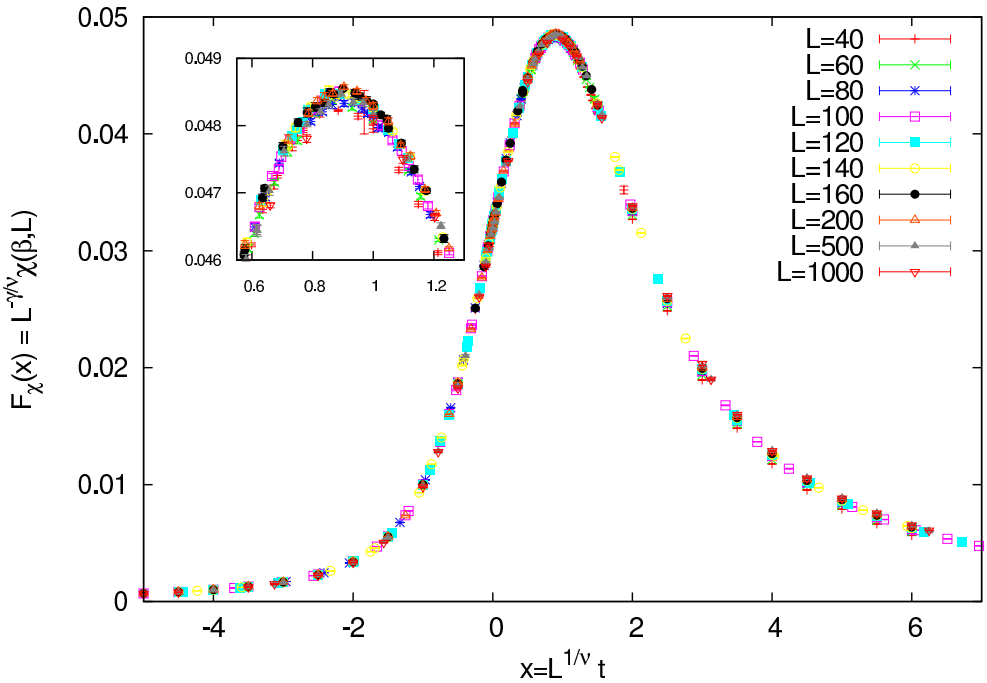



Figure 14.27: Collapse of the plots $\chi(\beta, L)$ for several values of $L$ according to equation (14.59). The known values $\beta_c = \ln(1+\sqrt{2})/2$, $\nu = 1$ and $\gamma/\nu = 7/4$ have been used.

result is

$$\begin{aligned} \beta_c &= 0.44069 \pm 0.00001 \\ \nu &= 1.00 \pm 0.01 \\ \frac{\gamma}{\nu} &= 1.750 \pm 0.002\,, \end{aligned}$$

Notice that, this crude estimate yields results whose accuracy is comparable to the previously calculated ones!

A similar procedure can be followed for other scaling observables, like the specific heat and the magnetization. Equations (14.8) and (14.9) generalize to[34]

$$\langle m \rangle(\beta, L) = L^{-\beta/\nu} F_m(L^{1/\nu} t)\,, \tag{14.60}$$

and

$$c(\beta, L) = L^{\alpha/\nu} F_c(L^{1/\nu} t) = \log(L) F_c(L^{1/\nu} t)\,, \tag{14.61}$$

since $\alpha = 0$. The results are shown in figures 14.28 and 14.29 respectively.

[34] In this relation $\beta$ on the left hand side is the temperature, whereas on the right hand side the critical exponent in (14.5).

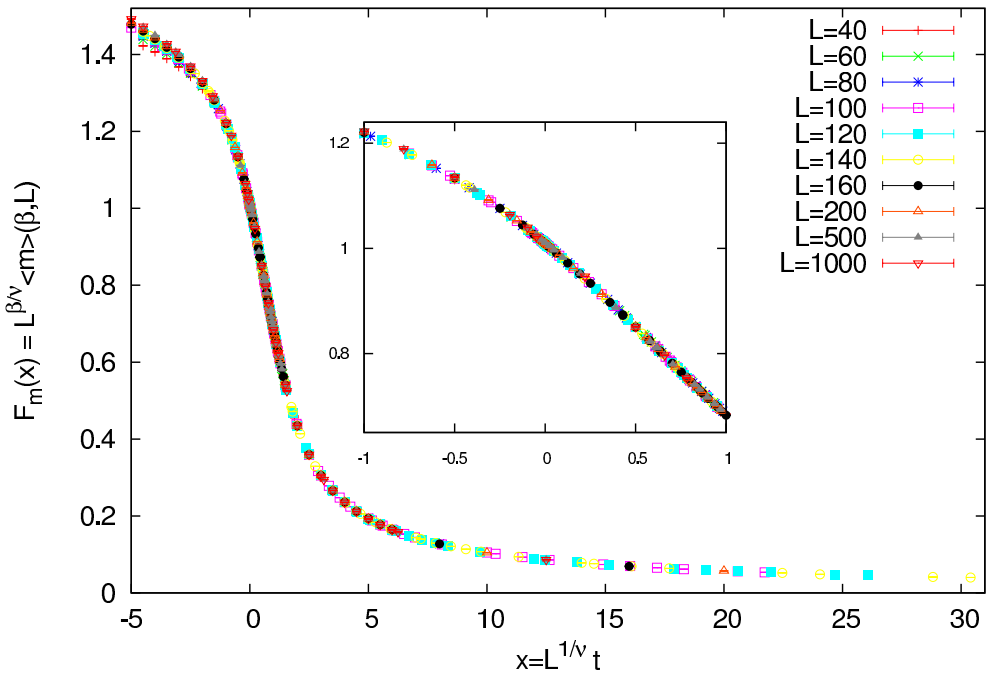



Figure 14.28: Collapse of the plots $\langle m\rangle(\beta, L)$ for several values of $L$ according to equation (14.60). The known values $\beta_c = \ln(1+\sqrt{2})/2$, $\nu = 1$ and $\beta/\nu = 1/8$ have been used.

Below, we list a `gnuplot` program in order to construct plots like the ones shown in figures 14.27–14.29. If we assume that the data are in a file named `all` in the following format[35]:

```
# ##########################################################
# e L beta <e>    +/- err c    +/- err
# m L beta <m>    +/- err chi +/- err
# n L beta <n>/N +/- err
# ----------------------------------------------------------
....
e 1000 0.462721 -0.79839031 7.506e-07 0.290266 0.00027
m 1000 0.462721  0.82648701 1.384e-06 2.137    0.00179
....
```

where the lines starting with the character `m` contain $(L, \beta, \langle m\rangle, \delta\langle m\rangle, \chi, \delta\chi)$ whereas the ones starting with `e` contain $(L, \beta, \langle e\rangle, \delta\langle e\rangle, c, \delta c)$. The program can be found in the file `scale_gamma.gpl`:

[35]This file can be found in the accompanying software, also named `all`.

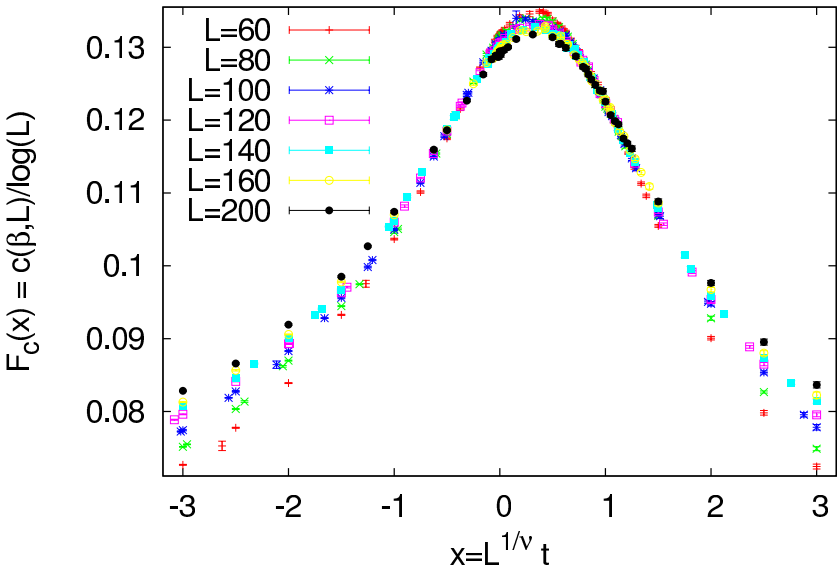



Figure 14.29: Collapse of the plots $c(\beta, L)$ for several values of $L$ according to equation (14.61). The known values $\beta_c = \ln(1+\sqrt{2})/2$ and $\nu = 1$ have been used.

```
# Usage:
# Ls = "40 60 80 100 120 140 160 200 500 1000"
# bc = bcc; nu = 1 ; gnu = 1.75; load "scale_gamma.gpl";
# Ls:  the values of L used in the collapse
# bc:  the critical temperature used in the calculation of
#      t=(beta_c-beta)/beta_c
# nu:  the exponent used in the calculation of x=L^{1/nu} t
# gnu: the exponent used in the calculation of
#      F_chi = L^{-gnu} chi(beta,L)

#the exact critical temperature (use bc=bcc is you wish):
bcc   = 0.5*log(1.0+sqrt(2.0));
NLs   = words(Ls);  # The number of lattice sizes
LL(i) = word (Ls,i);# Returns the i_th lattice size
cplot(i) = sprintf("\
   <grep 'm %s ' all|\
   sort -k 3,3g|\
   awk -v L=%s -v bc=%f -v nu=%f -v gnu=%f \
   '{print L^(1.0/nu)*(bc-$3)/bc,L^(-gnu)*$6,L^(-gnu)*$7}'\
                    ",LL(i),LL(i),bc,nu,gnu);

set macros
set term wxt enhanced

set title sprintf("b_c= %f nu= %f g/n= %f",bc,nu,gnu)
```

```
set xlabel "x=L^{1/nu} t"
set ylabel "F(x) = L^{-g/n} chi({/Symbol b},L)"

plot for[i=1:NLs] cplot(i) u 1:2:3 w e t sprintf("L=%s",LL(i))
```

In order to use the above program, we give the gnuplot commands

```
gnuplot> Ls = "40 60 80 100 120 140 160 200 500 1000"
gnuplot> bc = 0.4406868; nu = 1 ; gnu = 1.75;
gnuplot> load "scale_gamma.gpl"
```

The first two lines define the parameters of the plot. The variable `Ls` contains all the lattice sizes that we want to study, each value of $L$ separated from another by one or more spaces. The variables `bc, nu, gnu` are the parameters $\beta_c$, $\nu$ and $\gamma/\nu$ that will be used in the scaling relation (14.59). The third command calls the program that makes the plot. If we need to vary the parameters, then we redefine the corresponding variables and ... repeat.

In order to dissect the above program, look at the online help manual of gnuplot[36]. We will concentrate on the construction of the `awk` filter that computes the points in the plot properly normalized. The value of the function `cplot(i)` is a *string* of characters which varies with $L$ (the index `i` corresponds to the `i`-th word of the variable `Ls`). For each `i`, this string is substituted in the `plot` command, and it is of the form `"< grep ... L(-gnu)*$7}'"`. The values of the parameters are passed using the function `sprintf()`, which is called each time with a different value of `i`. The dirty work is done by `awk`, which calculates each point (columns 6 and 7: `$6, $7` are $\chi$ and its error $\delta\chi$). For *a given value* of $L$, the `grep` component of the filter prints the lines of the file `all` which contain the magnetization. The `sort` component sorts data in the order of increasing temperature (column 3: `-k3,3g`)

Can we make the above study more systematic and apply quantitative criteria for the quality of the collapse, which will help us estimate the error in the results? One crude way to estimate errors is to split the data in $n_b$ bins and work independently on each set of data. Each bin will give an optimal set of parameters $(\beta_c, \nu, \gamma/\nu)$ which will be assumed to be an independent measurement. The errors can be calculated using equation (13.39) (for $n = n_b$).

In order to provide a quantitative measure of the quality of the collapse, we define a $\chi^2$/dof similar to the one used in data fitting, as

[36] In order to look for help from gnuplot's online help system, use the commands `help word, help words, help macros, help sprintf, help plot iteration`

discussed in appendix 13.7. When the distance between the collapsing curves increases, this $\chi^2/\text{dof}$ should increase. Assume that our measurements consist of $N_L$ sets of measurements for $L = L_1, L_2, \ldots, L_{N_L}$. After setting the parameters $\mathbf{p} \equiv (\beta_c, \nu, \gamma/\nu)$ and an interval $\Delta x \equiv [x_{\text{min}}, x_{\text{max}}]$, we calculate the data sets $\{(x_{i,k}, F_\chi(x_{i,k}; \mathbf{p}, L_i))\}_{k=1,\ldots,n_i}$, for all $x_{i,k} \in \Delta X$, using our measurements. The data sets consist of $n_i$ points of the data for $L = L_i$, for which the $x_k$ are in the interval $\Delta x$. For each point $x_k$, we calculate the scaling function $F_\chi(x_{i,k}; \mathbf{p}, L_i) = L_i^{-\gamma/\nu}\chi(\beta_{i,k}, L_i)$, which depends on the chosen parameters $\mathbf{p}$ and the lattice size $L_i$. Then, we have to choose an interpolation method, which will define the interpolation *functions* $F_\chi(x; \mathbf{p}, L_i)$[37], so that we obtain a *good* estimate of the scaling function between two data points. Then, each point $\{(x_{i,k}, F_\chi(x_{i,k}; \mathbf{p}, L_i))\}_i$ in a data set has a well defined distance from every other data set $j$ ($j = 1, \ldots, N_L$ και $j \neq i$), which is defined by the distance from the interpolating function of the other sets. This is equal to[38] $|F_\chi(x_{i,k}; \mathbf{p}, L_i) - F_\chi(x_{i,k}; \mathbf{p}, L_j)|$. We define the quantity

$$\begin{aligned} \chi^2(\mathbf{p}; \Delta x) &= \frac{1}{N_L(N_L-1)n_{\text{points}}} \\ &\quad \times \sum_{i=1}^{N_L} \sum_{\{j=1, j\neq i\}}^{N_L} \sum_{k=1}^{n_i} \frac{\left(F_\chi(x_{i,k}; \mathbf{p}, L_i) - F_\chi(x_{i,k}; \mathbf{p}, L_j)\right)^2}{\left(\delta F_\chi(x_{i,k}; \mathbf{p}, L_i)\right)^2}, \end{aligned} \tag{14.62}$$

where $n_{\text{points}} = \sum_{i=1}^{N_L} n_i$ is the number of terms in the sum. The normalization constant $N_L(N_L-1)$ is used, because this is the number of pairs of curves in the sum. Each term is weighted by its error $\delta F_\chi(x_{i,k}; \mathbf{p}, L_i)$, so that points with small error have a larger contribution than points with large error. This is the definition used in [68], but you can see other approaches in [4], [66].

The $\chi^2(\mathbf{p}; \Delta x)$ depends on the parameters $\mathbf{p}$ and the interval $\Delta x$. Initially, we keep $\Delta x$ fixed and perform a minimization with respect to the parameters $\mathbf{p}$. The minimum is given by the values $\mathbf{p}_{\text{min}}$ and these values are the estimators that we are looking for. In order to calculate the errors $\delta\mathbf{p}$ we can bin our data according to the discussion on page 625. Alternatively, we may assume a $\chi^2$ distribution of the measurements, and if the minimum $\chi^2 \lesssim 1$, then the intervals of the parameter values that keep $\chi^2 \lesssim 2$ give an estimate of the errors in the parameters.

[37] This can be a polynomial interpolation, a cspline interpolation or one of its generalizations or multihistogramming. The last one is slightly more involved but carries smaller systematic errors.

[38] You can see the necessity of the interpolation, since the value $x_{i,k}$ most likely doesn't exist in the data set $j$.

The results depend on the chosen interval $\Delta x$. Usually [66], this is chosen so that its center is at the maximum of $F_\chi(x)$ so that $\Delta x = [x_{\max} - \delta x, x_{\max} + \delta x]$. If $\delta x$ is larger than it should, then $\chi^2(\mathbf{p}_{\min}; \Delta x)$ is large and we don't have good scaling. If it is too small, then the errors $\delta\mathbf{p}$ will be large. By taking the limit $\delta x \to 0$, we calculate $\mathbf{p}$ by studying the convergence of $\mathbf{p}_{\min}$ to stable, optimal with respect to error, values (see figure 8.7, page 238 in [4] as well as [66]).

## 14.11 Binder Cumulant

Up to now, we have studied fluctuations of observables by computing second order cumulants[39]. The calculation of the critical temperature, the order of the phase transition and the critical exponents can also be done by computing higher order cumulants and sometimes this calculation gives more accurate and clear results.The most famous one is the Binder cumulant which is a fourth order cumulant[40], and its name derives from Kurt Binder who studied it first [69,70],

$$U = 1 - \frac{\langle m^4 \rangle}{3\langle m^2 \rangle} \,. \tag{14.63}$$

Appendix 14.12 discusses its properties in detail. For a continuous phase transition

$$U = \begin{cases} 0 & \beta \ll \beta_c \\ U^* & \beta = \beta_c \\ \frac{2}{3} & \beta \gg \beta_c \end{cases} , \tag{14.64}$$

where for the Ising model on a square lattice $U^* = 0.610690(1)$ [70]. The value $U = 0$ corresponds to the Gaussian distribution, whereas the value $U = 2/3$ corresponds to two Gaussian distributions of small width around two symmetric values $\pm\langle m \rangle$ (see problems 14 and 15).

It practice, it is found that finite size corrections to $U^*$ are small, therefore the calculation of $U(\beta, L)$ gives an accurate measurement of the critical temperature $\beta_c$. The curves $U(\beta, L)$ intersect at the point ($\beta_c$, $U^*$) for different $L$ and this point gives a very good estimate of $\beta_c$.

Figure 14.30 shows our measurements for $U(\beta, L)$. The intersection of the curves in the figure at a single point ($\beta_c$, $U^*$) is impressively accurate.

[39] `http://en.wikipedia.org/wiki/Cumulant`, `http://mathworld.wolfram.com/Cumulant-GeneratingFunction.html`

[40] In statistics, the 4th order cumulant of a random variable $x$ is equal to $\kappa_4 = \langle (x - \langle x \rangle)^4 \rangle - 3\langle (x - \langle x \rangle)^2 \rangle$ and $\kappa_2 = \langle (x - \langle x \rangle)^2 \rangle$, so that $U = -\kappa_4/3\kappa_2$ for $x = m$ and $\langle m \rangle = 0$.

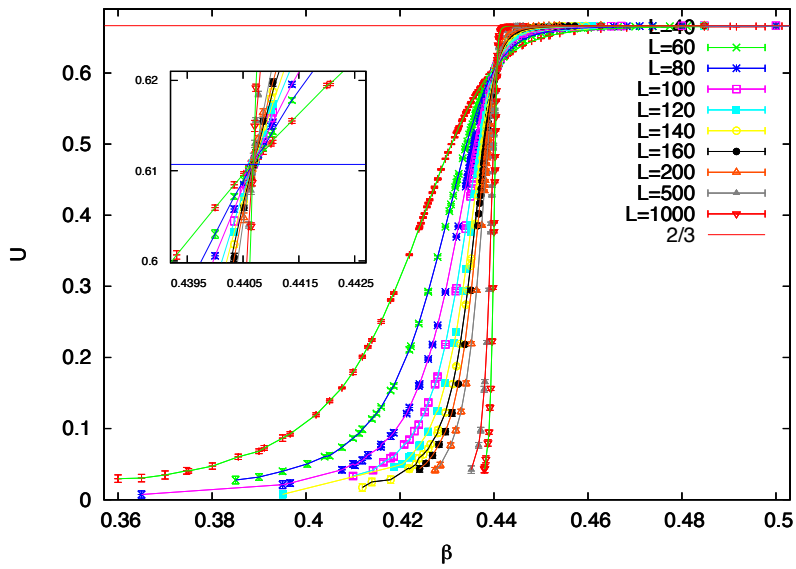



Figure 14.30: Binder cumulant for the Ising model on the square lattice for different temperatures and lattices sizes. The horizontal line is the expected value $U^* = 0.610690(1)$ [70].

Table 14.6 shows an attempt to calculate $\beta_c$ systematically by computing the critical temperature from the intersection of the curves $U(\beta, L)$ for three values of $L$. By taking into account all the measurements for $L = 100 - 1000$ the computed result is

$$\beta_c = 0.440678(9) \qquad U^* = 0.6107(4)\,, \tag{14.65}$$

which is in a very good agreement with the expected values $\beta_c = 0.44068679\ldots$, $U^* = 0.610690(1)$. Notice that, in the calculation of $U^*$ the systematic error due to finite size effects decreases with increasing $L$, whereas the statistical error increases due to the increase of the slope of the curves $U(\beta, L)$ near the point $\beta = \beta_c$. But the accuracy of the calculation of $\beta_c$ turns out to be better with increasing $L$.

Finite size scaling can also be applied to the Binder cumulant in order to calculate $\beta_c$ and $1/\nu$. From equation (14.90) (14.119) of appendix 14.12, we expect that $U$ scales as

$$U = F_U(x) = F_U(L^{1/\nu} t)\,. \tag{14.66}$$

This is confirmed in figure 14.31. From the value $F_U(x = 0)$, we obtain $U^* = 0.6107(4)$, which is consistent with the result (14.65).

| $L$ | | | $\beta_c$ | $U^*$ |
|---|---|---|---|---|
| 40 | 60 | 80 | 0.44069(4) | 0.6109(5) |
| 60 | 80 | 100 | 0.44069(4) | 0.6108(7) |
| 80 | 100 | 120 | 0.44068(4) | 0.6108(7) |
| 100 | 120 | 140 | 0.44069(4) | 0.6108(11) |
| 120 | 140 | 160 | 0.44069(4) | 0.6109(20) |
| 140 | 160 | 200 | 0.44067(3) | 0.6102(12) |
| 160 | 200 | 500 | 0.44067(2) | 0.6105(10) |
| 200 | 500 | 1000 | 0.44068(1) | 0.6106(9) |

Table 14.6: The calculation of $\beta_c$ and $U^*$ from the intersection of the curves $U(\beta, L)$ for fixed $L$ shown in figure 14.30. Each calculation uses three values of $L$. The expected values from the theory and the bibliography [70] are $\beta_c = 0.44068679\ldots$ and $U^* = 0.610690(1)$ respectively.

The numerical calculation of critical exponents, and especially $1/\nu$, can be hard in the general case. Therefore it is desirable to cross check results using several observables which have known scaling behavior. We discuss some of them below[41]. They involve the correlations of the magnetization with the energy.

The derivative of the Binder cumulant is

$$D_U = \frac{\partial U}{\partial \beta} = \frac{\langle m^4 E\rangle\langle m^2\rangle + \langle m^4\rangle\left(\langle m^2\rangle\langle E\rangle - 2\langle m^2 E\rangle\right)}{3\langle m^2\rangle^3}\,. \tag{14.67}$$

Its scaling is given by equation (14.120)

$$D_U = L^{1/\nu} F_{D_U}(x) = L^{1/\nu} F_{D_U}(L^{1/\nu} t)\,, \tag{14.68}$$

which us plotted in figure 14.32. Notice that $D_U$ defines a pseudocritical region around its maximum. The scaling of the maximum as well as the scaling of its position can be used in order to compute $1/\nu$, as we did in figures 14.23 and 14.25 for the magnetic susceptibility.

It could also turn out to be useful to study correlation functions of the form

$$D_{\ln m^n} = \frac{\partial \ln\langle m^n\rangle}{\partial \beta} = \langle E\rangle - \frac{\langle E m^n\rangle}{\langle m^n\rangle}\,, \tag{14.69}$$

whose scaling properties are given by equation (14.126) of appendix 14.12,

$$D_{\ln m^n} = L^{1/\nu} F_{D_{\ln m^n}}(x) = L^{1/\nu} F_{D_{\ln m^n}}(L^{1/\nu} t)\,. \tag{14.70}$$

[41] These have been particularly successful in the study of the 3d Ising model [72].

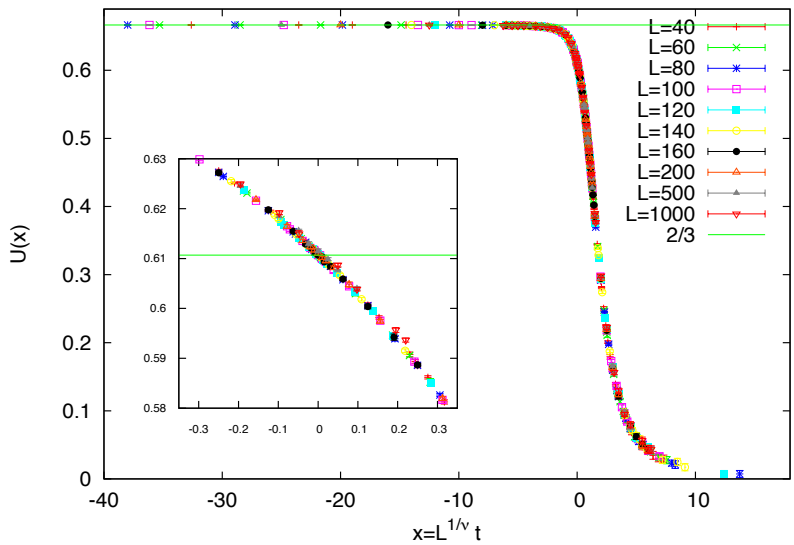



Figure 14.31: Scaling of the Binder cumulant for $1/\nu = 1$ and by using the exactly known critical temperature $\beta_c$ in $t = (\beta_c - \beta)/\beta_c$. The inset zooms in the critical region. The horizontal line is the expected result $U^* = 0.610690(1)$ [70].

In particular we are interested in the case $n = 1$

$$D_{\ln|m|} = \frac{\partial \ln\langle|m|\rangle}{\partial\beta} = \langle E\rangle - \frac{\langle E|m|\rangle}{\langle|m|\rangle}\,, \tag{14.71}$$

and $n = 2$

$$D_{\ln m^2} = \frac{\partial \ln\langle m^2\rangle}{\partial\beta} = \langle E\rangle - \frac{\langle Em^2\rangle}{\langle m^2\rangle}\,. \tag{14.72}$$

We also mention the energy cumulant $V$

$$V = 1 - \frac{\langle e^4\rangle}{3\langle e^2\rangle^2}\,. \tag{14.73}$$

In [73], it is shown that for a second order phase transition $V^* = 2/3$, whereas for a first order phase transition, we obtain a non trivial value. Therefore, this parameter can be used in order to determine whereas a system undergoes a first order phase transition. This is confirmed in figure 14.35. The minima of the curves $V(\beta, L)$ converge to the critical temperature according to (14.50).

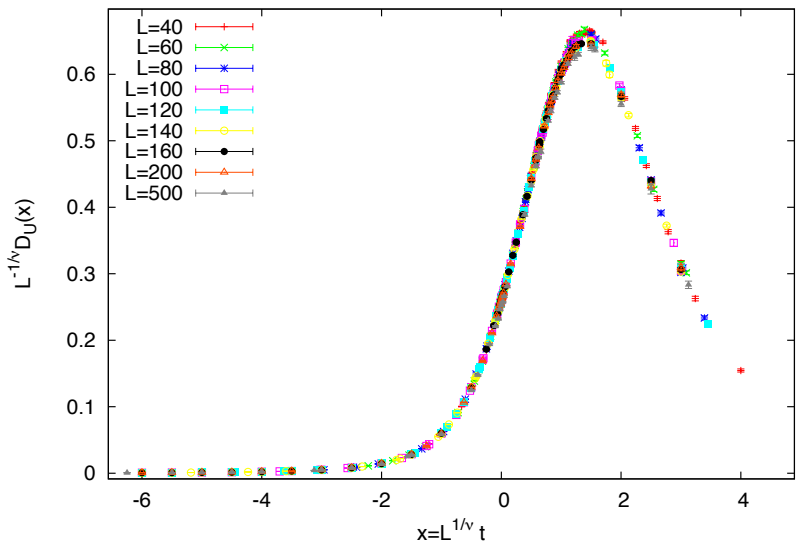



Figure 14.32: Scaling of the derivative of the Binder cumulant $D_U$ (see equation (14.67)) for $1/\nu = 1$ and $\beta_c$ equal to its known value in $t = (\beta_c - \beta)/\beta_c$.

## 14.12 Appendix: Scaling

### 14.12.1 Binder Cumulant

In section 14.11, we studied the scaling properties of the Binder cumulant

$$U = 1 - \frac{\langle m^4 \rangle}{3\langle m^2 \rangle^2} \tag{14.74}$$

numerically. In this appendix, we will use the general scaling properties of a system that undergoes a continuum phase transition near its critical temperature, in order to derive the scaling properties of $U$ and its derivatives. For more details, the reader is referred to [70], [6].

The values of $U$ are trivial in two cases: When the magnetization follows a Gaussian distribution, which is true in the high temperature, disordered phase, we have that $U = 0$. When we are in the low temperature, ordered phase, we have that $U = 2/3$. The proof is easy and it is left as an exercise (see problems 14 and 15).

According to the discussion in chapter 14, when the critical temperature $\beta_c$ of a continuum phase transition is approached, the system exhibits scaling properties due to the diverging correlation length $\xi$. If we approach $\beta_c$ from the high temperature phase, then we expect that

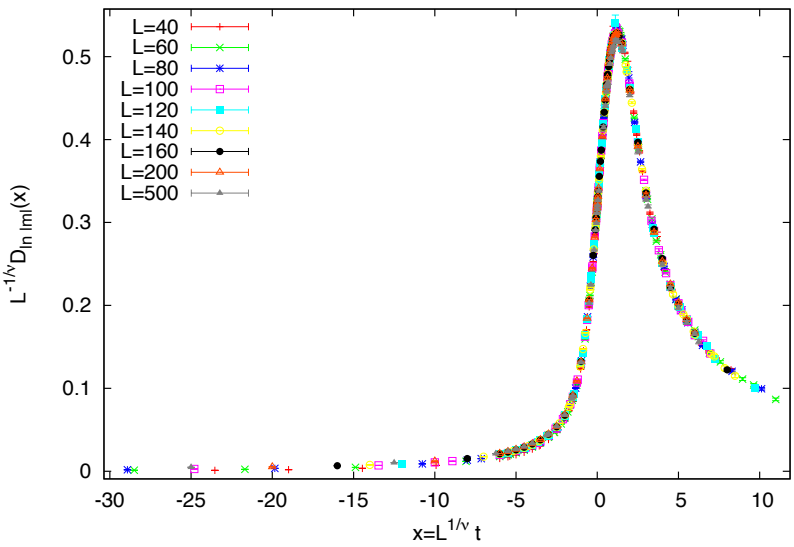



Figure 14.33: Scaling of $D_{\ln|m|}$ (see equation (14.71)) for $1/\nu = 1$ using the exact value of $\beta_c$ in $t = (\beta_c - \beta)/\beta_c$.

the distribution function of the magnetization per site $s$ (not its absolute value) is approximately of the form

$$P(L,s) = \frac{1}{\sqrt{2\pi\langle s^2\rangle}} \mathrm{e}^{-\frac{s^2}{2\langle s^2\rangle}} = \left(\frac{\beta L^d}{2\pi\chi}\right)^{\frac{1}{2}} \mathrm{e}^{-s^2\frac{L^d\beta}{2\chi}}\,, \tag{14.75}$$

which is a Gaussian with standard deviation $\sigma^2 = \langle s^2\rangle = \chi/(\beta L^d)$. We have temporarily assumed that the system is defined on a $d$–dimensional hypercubic lattice of edge $L$.

When the critical temperature is approached, the distribution function $P(L,s)$ scales according to the relation [70]

$$P(L,s) = L^x p_0 \tilde{P}(aL^y s, \frac{L}{\xi})\,, \tag{14.76}$$

where $\xi = \xi(t) = \lim_{L\to\infty}\xi(\beta,L)$, $t = (\beta_c - \beta)/\beta_c$, is the correlation length *in the thermodynamic limit*. As we approach the critical point, $\lim_{t\to 0}\xi(t) = +\infty$, in such a way that $\xi \sim |t|^{-\nu}$. Equation (14.76) is a *scaling hypothesis* which plays a fundamental role in the study of critical phenomena.

In order to calculate the exponents in equation (14.76), we apply the

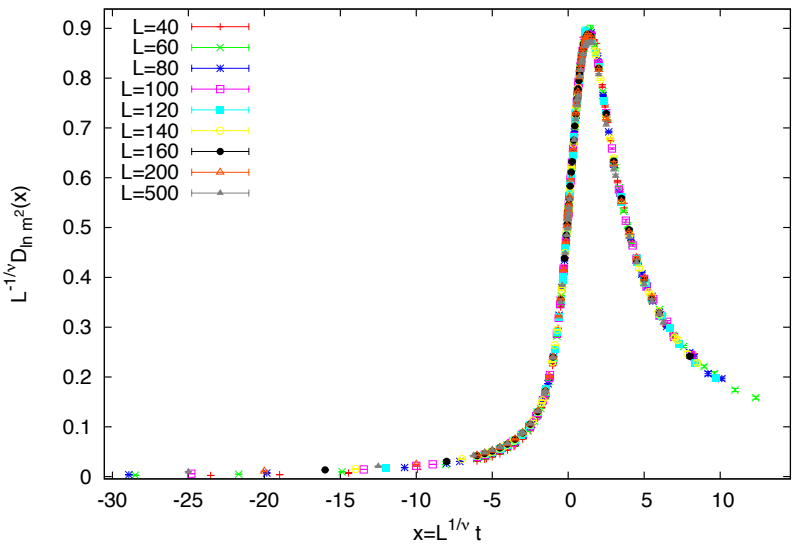



Figure 14.34: Scaling of $D_{\ln m^2}$ (see equation (14.72)) for $1/\nu = 1$ using the exact value of $\beta_c$ in $t = (\beta_c - \beta)/\beta_c$.

normalization condition of a probability distribution function

$$\begin{aligned} 1 &= \int_{\infty}^{+\infty} ds\, P(L,s) \\ &= L^x p_0 \int_{\infty}^{+\infty} ds\, \tilde{P}(aL^y s, \frac{L}{\xi}) \\ &= L^x p_0 \frac{1}{aL^y} \int_{\infty}^{+\infty} dz\, \tilde{P}(z, \frac{L}{\xi}) \\ &= L^{x-y} p_0 \frac{1}{a} \int_{\infty}^{+\infty} dz\, \tilde{P}(z, \frac{L}{\xi})\,, \end{aligned} \tag{14.77}$$

where we set $z = aL^y s$. For the left hand side to be equal to one, we must have that $x = y$,

$$C_0 \equiv \int_{\infty}^{+\infty} dz\, \tilde{P}(z, \frac{L}{\xi}) < \infty\,, \tag{14.78}$$

and $p_0 = a/C_0$. $C_0 = C_0(L/\xi)$, $p_0 = p_0(L/\xi)$ and $p_0 C_0 = a$ is a constant independent of $L$ and $\xi$. Finally, we obtain

$$P(L,s) = \frac{a}{C_0} L^y \tilde{P}(aL^y s, \frac{L}{\xi})\,. \tag{14.79}$$

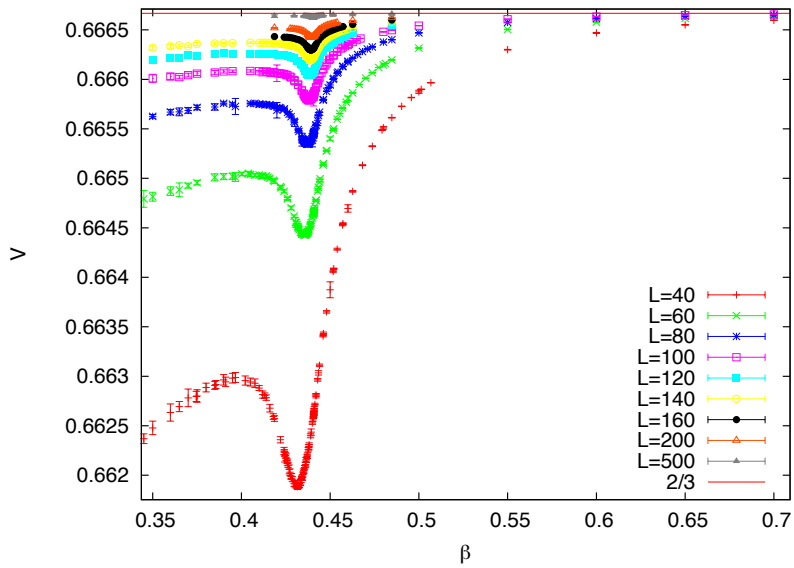



Figure 14.35: The energy cumulant defined by equation (14.73). As $L$ is increased, its value converges to $2/3$, as expected for a second order phase transition. The position of the minima converge to the critical temperature as $L^{-1/\nu}$

The moments of the distribution of the spins $\langle s^k \rangle$ are

$$\begin{aligned}
\langle s^k \rangle &= \int_{\infty}^{+\infty} ds\, s^k\, P(L,s) \\
&= \frac{a}{C_0} L^y \int_{\infty}^{+\infty} ds\, s^k\, \tilde{P}(aL^y s, \frac{L}{\xi}) \\
&= \frac{a}{C_0} L^y \frac{1}{a^{k+1} L^{(k+1)y}} \int_{\infty}^{+\infty} dz\, z^k\, \tilde{P}(z, \frac{L}{\xi}) \\
&= L^{-ky} F_k\left(\frac{L}{\xi}\right) , \qquad (14.80)
\end{aligned}$$

where the last line is the definition of the function $F_k(x)$. When we *first* take the thermodynamic limit $L \to \infty$ and *then* approach the critical temperature $t \to 0$, the correlation length $\xi \to \infty$ diverges in such a way that $L/\xi \to \infty$. In the region $\beta < \beta_c$ ($\langle m \rangle = 0$) we have that $\chi = \beta L^d \langle s^2 \rangle$, and by using the relations

$$\left.\begin{aligned} \chi &= \chi_+ t^{-\gamma} \\ \xi &= \xi_+ t^{-\nu} \end{aligned}\right\} \Rightarrow \chi = \frac{\chi_+}{\xi_+^{\gamma/\nu}} \xi^{\gamma/\nu} \sim \xi^{\gamma/\nu} , \qquad (14.81)$$

we obtain

$$\langle s^2 \rangle = \beta^{-1} L^{-d} \chi = \frac{\chi^+}{\beta L^d \xi_+^{\gamma/\nu}} \, \xi^{\gamma/\nu} \sim \xi^{\gamma/\nu} \, . \tag{14.82}$$

In the above equations, we introduced the universal amplitudes $\chi_+$ και $\xi_+$, which are universal constants (i.e. they are the same within a universality class) and they are defined from equation (14.81). In this limit, in order for (14.80) to have consistent scaling for $k = 2$ on the right and left hand sides[42], we obtain (compare with equation (14.57))

$$F_2 \left( \frac{L}{\xi} \right) \sim \left( \frac{L}{\xi} \right)^{-\gamma/\nu} \qquad \text{for} \qquad \frac{L}{\xi} \gg 1 \, . \tag{14.83}$$

In order to compute the $L$-scaling, we substitute the above equations to (14.80) for $k = 2$, and we obtain

$$\frac{\chi^+}{\beta L^d \xi_+^{\gamma/\nu}} \, \xi^{\gamma/\nu} \sim L^{-2y} \left( \frac{L}{\xi} \right)^{-\gamma/\nu} \, . \tag{14.84}$$

Then, we obtain

$$L^{-d} \sim L^{-2y} \, L^{-\gamma/\nu} \Rightarrow d = 2y + \frac{\gamma}{\nu} \Rightarrow y = \frac{d\nu - \gamma}{2\nu} = \frac{\beta}{\nu} \, , \tag{14.85}$$

where we used the known[43] hyperscaling relation

$$d \, \nu = \gamma + 2\beta \, . \tag{14.86}$$

Finally, we obtain the equations $\beta < \beta_c$,

$$P(L, s) = \frac{a}{C_0} L^{\beta/\nu} \tilde{P}(a L^{\beta/\nu} s, \frac{L}{\xi}) \, , \tag{14.87}$$

$$\langle s^2 \rangle = L^{-2\beta/\nu} F_2 \left( \frac{L}{\xi} \right) \, , \tag{14.88}$$

$$\langle s^4 \rangle = L^{-4\beta/\nu} F_4 \left( \frac{L}{\xi} \right) \, , \tag{14.89}$$

which are valid in the disordered phase $\beta < \beta_c$. From equation (14.74), we find that the critical behavior of the Binder cumulant is

$$U \sim 1 - \frac{L^{-4\beta/\nu} F_4 \left( \frac{L}{\xi} \right)}{3 L^{-4\beta/\nu} F_2 \left( \frac{L}{\xi} \right)^2} = 1 - \frac{1}{3} \frac{F_4 \left( \frac{L}{\xi} \right)}{F_2 \left( \frac{L}{\xi} \right)^2} \, . \tag{14.90}$$

[42] i.e. both sides should scale w.r.t to the correlation length as $\sim \xi^{\gamma/\nu}$.

[43] See e.g. [71], equations 3.35, 3.36, 3.53.

Finite size effects dominate in the pseudocritical region, in which case we take the thermodynamic limit $L \to \infty$ keeping $L/\xi$ finite, and the fluctuations get suppressed, rendering the functions $F_k(x)$ finite. Therefore, we obtain[44]

$$\lim_{L\to\infty} U(t=0,L) \equiv U^* = 1 - \frac{1}{3}\frac{F_4(0)}{F_2(0)^2} = \text{const.}\,, \tag{14.91}$$

which shows why the value of $U$ at the critical temperature turned out to be almost independent of the system size $L$. $U^*$ is found to depend on the boundary conditions and on the anisotropy of the interaction. For the Ising model on the square lattice we have that [70] (Kamieniarz+Blöte)

$$U^* = 0.610690(1) \tag{14.92}$$

### 14.12.2 Scaling

Consider a change of length scale on a lattice so that

$$\xi \to \frac{\xi}{b}\,, \tag{14.93}$$

where $\xi$ is the dimensionless correlation length in the thermodynamic limit and $b$ is the scaling factor. Then, the basic assumption for the scaling of thermodynamics quantities in the region of a continuous phase transition is that the free energy is changed according to [45]

$$f(t,h) = b^{-d} f(tb^{y_t}, hb^{y_h})\,, \tag{14.94}$$

where $t$ is the reduced temperature and $h$ is the external magnetic field[46]. The above relation summarizes the *scaling hypothesis*, and it is a relation similar to (14.76). This relation can be understood through the renormalization group approach, and the fundamental assumption is the appearance of a unique dynamical length scale that diverges as we approach the critical point. The arguments $tb^{y_t}$ and $hb^{y_h}$ give the change in the coupling constants $t$ and $h$ under the change in length scale in order that the equation remains valid.

By applying the above relation $n$ times we obtain

$$f(t,h) = b^{-nd} f(tb^{ny_t}, hb^{ny_h})\,. \tag{14.95}$$

[44] At $t=0$ we have $\xi(0) = +\infty$, therefore for finite $L$ we have that $L/\xi = 0$.
[45] More precisely the *singular part* of the free energy.
[46] See e.g. chapter 3 in [71].

If we take $n \to \infty$, $t \to 0$, keeping the product $tb^{ny_t} = t_0 = \mathcal{O}(1)$ fixed, we obtain

$$\begin{aligned} f(t,h) &= t^{d/y_t} f(t_0, ht^{-y_h/y_t}) \\ &\equiv t^{d/y_t} \Psi(ht^{-y_h/y_t}) \\ &= t^{2-\alpha} \Psi(ht^{-y_h/y_t}) \,, \end{aligned} \tag{14.96}$$

where we substituted $b^n \sim t^{-1/y_t}$ and *defined* the scaling function $\Psi(z)$ and the critical exponent

$$\alpha = 2 - \frac{d}{y_t} \,. \tag{14.97}$$

By applying the same reasoning to (14.93) for the correlation length we obtains

$$\xi(t,h) = b^{-1}\xi(tb^{y_t}, hb^{y_h}) = \ldots = b^{-n}\xi(tb^{ny_t}, hb^{ny_h}) \,. \tag{14.98}$$

By taking the limit $n \to \infty$, $t \to 0$, keeping the product $tb^{ny_t} \sim \mathcal{O}(1)$, the left hand side will give a finite value, e.g. $\xi_0 < \infty$ whereas the right hand side will give

$$\xi_0 = t^{1/y_t} \xi(t_0, ht^{-y_h/y_t}) \,. \tag{14.99}$$

By considering the case $h = 0$, and by comparing to the known relation (14.4) $\xi \sim t^{-\nu}$, we obtain

$$\xi = \xi_0 t^{-1/y_t} \Rightarrow \nu = \frac{1}{y_t} \,. \tag{14.100}$$

By taking the derivative of (14.96) with respect to the temperature, we obtain

$$\frac{\partial f}{\partial t} \sim t^{1-\alpha}\Psi(ht^{-y_h/y_t}) + t^{2-\alpha} h t^{-y_h/y_t - 1} \Psi'(ht^{-y_h/y_t}) \,. \tag{14.101}$$

We use the notation $\sim$ whenever we neglect terms that are not related to the scaling properties of a function.

By taking the derivative once more, and by setting $h = 0$, we obtain the specific heat

$$c \sim \frac{\partial^2 f}{\partial t^2} \sim t^{-\alpha}\Psi(0) \,. \tag{14.102}$$

Therefore, the critical exponent $\alpha$ is nothing but the critical exponent of the specific heat defined in equation (14.4).

The magnetic susceptibility can be obtained in a similar way by taking the derivative of (14.96) with respect to $h$

$$\frac{\partial f}{\partial h} \sim t^{d/y_t} t^{-y_h/y_t} \Psi'(ht^{-y_h/y_t}) \sim t^{\nu d - \nu y_h} \Psi'(ht^{-\nu y_h}) \,. \tag{14.103}$$

By taking the derivative once more time and by setting $h = 0$ we obtain the magnetic susceptibility

$$\chi \sim \frac{\partial^2 f}{\partial h^2} \sim t^{\nu d - 2\nu y_h} \Psi'(0)\,, \tag{14.104}$$

and, by comparing to (14.3) $\chi \sim t^{-\gamma}$, we obtain

$$\gamma = 2\nu y_h - \nu d \Leftrightarrow y_h = \frac{1}{2}(d + \frac{\gamma}{\nu}) = d - \frac{\beta}{\nu} = \frac{\beta + \gamma}{\nu} \tag{14.105}$$

In the last two equations we used the hyperscaling relations

$$\nu d = \gamma + 2\beta\,. \tag{14.106}$$

### 14.12.3 Finite Size Scaling

We will now extend the analysis of the previous section to the case of a system of finite size. We will assume that the system's degrees of freedom are located on a lattice whose linear size is $l = La$ (the volume is $V = l^d$, $d$ is the number of dimensions), where $L$ is the (dimensionless) number of lattice sites and $a$ is the lattice constant. We consider the limit $L \to \infty$ and $a \to 0$, so that $l$ *remains constant*. By changing the $L$-scale

$$L \to \frac{L}{b} \Leftrightarrow L^{-1} \to bL^{-1}\,, \tag{14.107}$$

and

$$a \to ba\,, \tag{14.108}$$

equation (14.94) generalizes to

$$f(t, h, L^{-1}) = b^{-nd} f(tb^{ny_t}, hL^{ny_h}, b^n L^{-1})\,. \tag{14.109}$$

By taking the limit $t \to 0$, $n \to \infty$ and $tb^{ny_t} = t_0 < \infty \Rightarrow b^n \sim t^{-1/y_t}$ (approach of the critical point), the above relation becomes

$$f(t, h, L^{-1}) = t^{d/y_t} f(t_0, ht^{-y_h/y_t}, t^{-1/y_t} L^{-1}) = t^{d/y_t} \Psi(ht^{-y_h/y_t}, t^{-1/y_t} L^{-1})\,. \tag{14.110}$$

By differentiating and setting $h = 0$ as in the previous section we obtain[47]

$$\chi(t, L^{-1}) = \left.\frac{\partial^2 f}{\partial h^2}\right|_{h=0} = t^{-\gamma} \phi_2(L^{-1} t^{-\nu}) = t^{-\gamma} \phi_2(\frac{\xi}{L})\,, \tag{14.111}$$

[47]We stress again that $\xi \sim t^{-\nu}$ in (14.111) is the correlation length in the thermodynamic limit and not at finite $L$.

where we set $y_t = 1/\nu$, $\phi_2(x) = \Psi^{(2,0)}(0,x) = \partial^2\Psi(z,x)/\partial z^2|_{z=0}$.

The thermodynamic limit is obtained for $L \gg \xi$ where $\phi_2(\frac{\xi}{L}) \to \phi_2(0) < \infty$, which yields the known relation $\chi \sim t^{-\gamma}$.

When $L$ is comparable to $\xi$, finite size effects dominate. The large fluctuations are suppressed and the magnetic susceptibility has a maximum at a crossover (pseudocritical) temperature $t_X \equiv (\beta_c - \beta_c(L))/\beta_c$, where $t_X \sim L^{-1/\nu}$. The last relation holds because $L \sim \xi \sim t^{-\nu}$ by assumption. We obtain

$$\chi_{\max} \sim t_X^{-\gamma}\phi_2(L^{-1}t_X^{-\nu}) \sim L^{\gamma/\nu}\phi_2(L^{-1}L) \sim L^{\gamma/\nu}\phi_2(1) \sim L^{\gamma/\nu}\,. \tag{14.112}$$

In the region of the maximum, we obtain the functional form

$$\chi(t, L^{-1}) = L^{\gamma/\nu}F_\chi(L^{1/\nu}t)\,, \tag{14.113}$$

which is nothing but equation (14.59). The function $F_\chi(x)$ is analytic in its argument $x = L^{1/\nu}t$, since for a finite system $\chi(t, L^{-1})$ is an analytic function of the temperature[48]. In the thermodynamic limit ($L \to \infty$ and $|t| > 0$, therefore $x \to \infty$)

$$F_\chi(x) \sim x^{-\gamma} \qquad x \gg 1\,, \tag{14.114}$$

so that $\chi(t, L^{-1}) = L^{\gamma/\nu}F_\chi(L^{1/\nu}t) \sim L^{\gamma/\nu}\,(L^{1/\nu}t)^\gamma \sim t^{-\gamma}$. Near the pseudocritical point

$$F_\chi(x) = F_{\chi,0} + F_{\chi,1}x + F_{\chi,2}x^2 + \ldots \qquad x \ll 1\,, \tag{14.115}$$

and we expect that for $L^{1/\nu}t \ll 1$ we have that

$$\chi(t, L^{-1}) = L^{\gamma/\nu}\left(1 + \chi_1 L^{1/\nu}t + \chi_2 L^{2/\nu}t^2 + \ldots\right)\,. \tag{14.116}$$

The above relations lead to the following conclusions:

- The pseudocritical point shifts as $\sim L^{-1/\nu}$ (equation (14.50))
- The peak of the magnetic susceptibility increases as $\chi_{\max} \sim L^{\gamma/\nu}$
- The direction of the shifting of the maximum of the magnetic susceptibility depends on the boundary conditions:

[48]This is because the partition function is an analytic function of the temperature. Therefore it is $x = L^{1/\nu}t$ which is the scaling variable and not a power of it, such as $\tilde{x}$ used in (14.53) and (14.54).

- Periodic boundary conditions suppress the effects of the fluctuations, since the wave vectors are limited by $\frac{2\pi}{L}n$. This increases the pseudocritical temperature $T_c(L)$ ($\beta_c(L) < \beta_c \Rightarrow c > 0$ in (14.50)).
- Free boundary conditions lead to free fluctuations on the boundary, which decrease the pseudocritical temperature $T_c(L)$ ($\beta_c(L) > \beta_c \Rightarrow c < 0$ in (14.50))
- Frozen (fixed) spins on the boundary lead to increased order in the system. This increases the pseudocritical temperature $T_c(L)$ ($\beta_c(L) < \beta_c \Rightarrow c > 0$ in (14.50)).

We conclude that $F_\chi(L^{1/\nu}t)$ depends on the boundary conditions and the geometry of the lattice.

Similarly, we obtain

$$\langle m^k \rangle \sim L^{-d}\frac{\partial^k f}{\partial h^k} \sim L^{-d}t^{d/y_t - ky_h/y_t}\phi_k(L^{-1}t^{-\nu}) \sim L^{-d}t^{\nu d - k\nu y_h}\phi_k\left(\frac{\xi}{L}\right), \tag{14.117}$$

and by following similar arguments leading to (14.113), we obtain

$$\langle m^k \rangle \sim L^{-d}L^{-d+ky_h}F_k(L^{1/\nu}t) \sim L^{k\frac{\beta+\gamma}{\nu}}F_k(L^{1/\nu}t)\,. \tag{14.118}$$

For the Binder cumulant we obtain

$$U = 1 - \frac{\langle m^4 \rangle}{3\langle m^2 \rangle^2} \sim 1 - \frac{L^{4y_h}F_4(L^{1/\nu}t)}{3(L^{2y_h}F_2(L^{1/\nu}t))^2} \sim U_* + U_1\cdot(L^{1/\nu}t) + U_2\cdot(L^{1/\nu}t)^2 + \dots\,, \tag{14.119}$$

where in the last equality we expanded the analytic functions $F_{2,4}(L^{1/\nu}t)$ for small $L^{1/\nu}t$. Then, we see that

$$\frac{\partial U}{\partial \beta} \sim \partial_t U \sim L^{1/\nu}\,. \tag{14.120}$$

By differentiating (14.110) with respect to the temperature we obtain

$$\begin{aligned}
\frac{\partial f}{\partial t} &\sim t^{d/y_t - 1}\Psi(ht^{y_h/y_t}, L^{-1}t^{-1/y_t}) \\
&\quad + t^{d/y_t}(ht^{y_h/y_t - 1})\Psi^{(1,0)}(ht^{y_h/y_t}, L^{-1}t^{-1/y_t}) \\
&\quad + t^{d/y_t}L^{-1}t^{-1/y_t - 1}\Psi^{(0,1)}(ht^{y_h/y_t}, L^{-1}t^{-1/y_t}) \\
&\sim t^{\nu d - 1}\Psi(ht^{\nu y_h}, L^{-1}t^{-\nu}) \\
&\quad + ht^{\nu d + \nu y_h - 1}\Psi^{(1,0)}(ht^{\nu y_h}, L^{-1}t^{-\nu}) \\
&\quad + L^{-1}t^{\nu d - 1 - \nu}\Psi^{(0,1)}(ht^{\nu y_h}, L^{-1}t^{-\nu})\,,
\end{aligned} \tag{14.121}$$

where we used the notation $\Psi^{(n,m)}(x,z) = \partial^{n+m}\Psi(x,z)/\partial x^n \partial z^m$. The term proportional to $h$ vanishes when we set $h=0$. In the pseudocritical region, where $t_X \sim L^{-1/\nu}$, the first and third term are of the same order in $L$ and we obtain

$$\left.\frac{\partial f}{\partial t}\right|_{h=0} = L^{-d+\frac{1}{\nu}} F^1(L^{1/\nu} t)\,, \tag{14.122}$$

and by successive differentiation

$$\left.\frac{\partial^k f}{\partial t^k}\right|_{h=0} = L^{-d+\frac{k}{\nu}} F^k(L^{1/\nu} t)\,. \tag{14.123}$$

The derivatives

$$\left.\frac{\partial^2 f}{\partial t \partial h}\right|_{h=0} = L^{-d+y_h+\frac{1}{\nu}} F_1^1(L^{1/\nu} t) = L^{\frac{1-\beta}{\nu}} F_1^1(L^{1/\nu} t)\,, \tag{14.124}$$

$$\left.\frac{\partial^{1+k} f}{\partial t \partial h^k}\right|_{h=0} = L^{-d+ky_h+\frac{1}{\nu}} F_k^1(L^{1/\nu} t)\,. \tag{14.125}$$

In particular

$$\frac{\langle E\, m^k\rangle}{\langle m^k\rangle} = \frac{\left.\frac{\partial^{1+k} f}{\partial t \partial h^k}\right|_{h=0}}{\left.\frac{\partial^k f}{\partial h^k}\right|_{h=0}} \sim \frac{L^{-d+ky_h+\frac{1}{\nu}}}{L^{-d+ky_h}} \sim L^{1/\nu} \tag{14.126}$$

$$\frac{\langle e^4\rangle}{\langle e^2\rangle^2} = \frac{L^{-d}\left.\frac{\partial^4 f}{\partial t^4}\right|_{h=0}}{\left(L^{-d}\left.\frac{\partial^2 f}{\partial t^2}\right|_{h=0}\right)^2} \sim \frac{L^{-\frac{4}{\nu}}}{(L^{-\frac{2}{\nu}})^2} \sim \text{const.} \tag{14.127}$$

## 14.13 Appendix: Critical Exponents

### 14.13.1 Definitions

$$\begin{array}{lll}
\alpha: & c \sim t^{-\alpha}, & c_{\max} \sim L^{\alpha/\nu}, \quad c(t,L) = L^{\alpha/\nu} F^2(L^{1/\nu} t) \\
\beta: & m \sim t^{\beta}, & m \sim L^{-\beta/\nu}, \quad m(t,L) = L^{-\beta/\nu} F_1(L^{1/\nu} t) \\
\gamma: & \chi \sim t^{\gamma}, & \chi_{\max} \sim L^{\gamma/\nu}, \quad \chi(t,L) = L^{\gamma/\nu} F_2(L^{1/\nu} t) \\
\nu: & \xi \sim t^{-\nu}, & \xi \sim L, \\
\delta: & M \sim h^{1/\delta} & \\
z: & \tau \sim \xi^z &
\end{array} \tag{14.128}$$

The scaling relation

$$f(t,h) = t^{d/y_t} \Psi(h t^{y_h/y_t})\,, \tag{14.129}$$

defines the exponents $y_t$, $y_h$. The relation

$$G(\mathbf{r}, t = 0) \sim \frac{1}{r^{d-2+\eta}}\,, \tag{14.130}$$

defines the exponent $\eta$ coming from the two point correlation function $G(\mathbf{r}, t) = \langle \mathbf{s}(\mathbf{r}) \cdot \mathbf{s}(\mathbf{0}) \rangle$.

### 14.13.2 Hyperscaling Relations

From the definitions and the hyperscaling relations we have that

$$\begin{aligned} \alpha + 2\beta + \gamma &= 2 \\ \gamma + 2\beta &= \nu d \\ 2 - \nu d &= \alpha \\ \alpha + \beta(1+\delta) &= 2 \\ \nu(2-\eta) &= \gamma \end{aligned} \tag{14.131}$$

$$y_t = \frac{1}{\nu} = \frac{d}{2-\alpha} \qquad y_h = \frac{\beta+\gamma}{\nu} = \frac{1}{2}\left(d + \frac{\gamma}{\nu}\right) = d - \frac{\beta}{\nu} \tag{14.132}$$

$$\alpha = 2 - \frac{d}{y_t} \qquad \beta = \frac{d - y_h}{y_t} \qquad \gamma = \frac{2y_h - d}{y_t} \qquad \delta = \frac{y_h}{d - y_h} \tag{14.133}$$

$$\eta = d + 2 - 2y_h \Leftrightarrow d - 2 + \eta = 2(d - y_h) \tag{14.134}$$

## 14.14 Problems

The files `all` and `allem` in the accompanying software contain measurements that you can use for your data analysis or compare them with your own measurements.

1. Compute the average acceptance ratio $\bar{A}$ for the Metropolis algorithm as a function of the temperature for $L = 10, 40, 100$. Compute the average size $\langle n \rangle$ of the Wolff clusters at the same values of the temperatures. Then calculate the number of Wolff clusters that are equivalent to a Metropolis sweep. Make the plots of all of your results and connect the points corresponding to the same $L$.

| Model | $\nu$ | $\alpha$ | $\beta$ | $\gamma$ | $\delta$ | $\eta$ | $y_t$ | $y_h$ |
|---|---|---|---|---|---|---|---|---|
| q=0 Potts (2d) [63] | $\infty$ | $-\infty$ | $\frac{1}{6}$ | $\infty$ | $\infty$ | 0 | 0 | 2 |
| q=1 Potts (2d) [63] | $\frac{4}{3}$ | $-\frac{2}{3}$ | $\frac{5}{36}$ | $2\frac{7}{18}$ | $18\frac{1}{5}$ | $\frac{5}{24}$ | $\frac{3}{4}$ | $\frac{91}{48}$ |
| Ising (2d) [63] | 1 | 0 | $\frac{1}{8}$ | $\frac{7}{4}$ | 15 | $\frac{1}{4}$ | 1 | $\frac{15}{8}$ |
| q=3 Potts (2d) [63] | $\frac{5}{6}$ | $\frac{1}{3}$ | $\frac{1}{9}$ | $\frac{13}{9}$ | 14 | $\frac{4}{15}$ | $\frac{6}{5}$ | $\frac{28}{15}$ |
| q=4 Potts (2d) [63,75] | $\frac{2}{3}$ | $\frac{2}{3}$ | $\frac{1}{12}$ | $\frac{7}{6}$ | 15 | $\frac{1}{4}$ | $\frac{3}{2}$ | $\frac{15}{8}$ |
| classical (4d) [74] | $\frac{1}{2}$ | 0 | $\frac{1}{2}$ | 1 | 3 | 0 | 2 | 3 |
| Spherical (3d) [74] | 1 | $-1$ | $\frac{1}{2}$ | 2 | 5 | 0 | 1 | $\frac{5}{2}$ |
| Ising (3d) [74] | — | $\frac{1}{8}$ | $\frac{5}{16}$ | $\frac{5}{4}$ | 5 | — | — | — |
| Ising (3d) [78] | 0.631 | 0.108(5) | 0.327(4) | 1.237(4) | 4.77(5) | 0.039 | — | — |
| Heisenberg (3d) [76] | 0.70 | $-0.1$ | 0.36 | 1.4 | 5 | 0.03 | — | — |
| XY (3d) [77] | 0.663 | — | — | 1.327(8) | — | — | — | — |
| AF q=3 Potts (3d) [79] | 0.66 | $-0.011$ | 0.351 | 1.309 | 4.73 | — | — | — |

Table 14.7: Critical exponents of the models referred to in the first column. Whenever the value is shown as a floating point number, the exponents are approximate. For the approximate values we don't apply the hyperscaling relations, but we simply mention the values reported in the bibliography. The values for the 3d Ising model in [74] are a conjecture. For the 3d Ising see also [40] p. 244. 3d XY and 3d AF q=3 Potts are conjectured to belong to the same universality class.

2. Make the plots in figures 14.6–14.10 and add data for $L = 50$, 120, 140, 160, 180, 200.

3. Make the plots in figures 14.11–14.12 and add data for $L = 50$, 90, 130, 150, 190, 250. Recalculate the dynamic exponent $z$ using your data.

4. Make the plot in figure 14.13 and add data for $L = 30$, 50, 70, 90. Recalculate the dynamic exponent $z$ using your data.

5. Reproduce the results shown in table 14.1. Add a 6th column computing $\tau^A_{m,\text{Metropolis}} = \tau_{m,\text{Metropolis}}\bar{A}$, where $\bar{A}$ is the average acceptance ratio of the Metropolis algorithm. This changes the unit of time to $N$ *accepted* spin flips. These are the numbers that are directly comparable with $\tau_m$.

6. Simulate the 2d Ising model on the square lattice for $L = 10$, 20, 40, 80, 100. Choose appropriate values of $\beta$, so that you will be able to determine the magnetic susceptibility and the specific heat with an accuracy comparable to the one shown in table 14.5. In each case, check for the thermalization of the system and calculate the errors.

| $L$ | $\chi(\beta_c, L)$ | | $\langle m\rangle(\beta_c, L)$ | | $c(\beta_c, L)$ | |
|---|---|---|---|---|---|---|
| 40 | 20.50 | 0.02 | 0.6364 | 0.0001 | 0.4883 | 0.0007 |
| 60 | 41.78 | 0.08 | 0.6049 | 0.0002 | 0.5390 | 0.0008 |
| 80 | 69.15 | 0.09 | 0.5835 | 0.0001 | 0.5743 | 0.0012 |
| 100 | 102.21 | 0.25 | 0.5673 | 0.0002 | 0.6026 | 0.0014 |
| 120 | 140.18 | 0.11 | 0.5548 | 0.0001 | 0.6235 | 0.0010 |
| 140 | 183.95 | 0.33 | 0.5442 | 0.0002 | 0.6434 | 0.0006 |
| 160 | 232.93 | 0.55 | 0.5351 | 0.0001 | 0.6584 | 0.0020 |
| 200 | 342.13 | 0.72 | 0.5206 | 0.0001 | 0.6858 | 0.0014 |
| 500 | 1687.2 | 4.4 | 0.4647 | 0.0002 | 0.7794 | 0.0018 |
| 1000 | 6245 | 664 | 0.4228 | 0.0040 | – | – |

Table 14.8: $\chi(\beta_c, L)$, $\langle m\rangle(\beta_c, L)$ and $c(\beta_c, L)$ at the critical temperature for different $L$ used in problem 9.

7. Make the fits that lead to the results (14.38), (14.39), (14.41) and (14.43)

8. Study the scaling of the specific heat as a function of the temperature. Compare the quality of the fits to the functions $a \log|t|$ and $a\,|t|^\alpha$ by computing the $\chi^2$/dof according to the discussion in appendix 13.7 after page 548.

9. Consider the table 14.8 showing the measurements of $\chi(\beta_c, L)$, $\langle m\rangle(\beta_c, L)$ and $c(\beta_c, L)$. Use the values in this table in order to make the fits which give the exponents $\gamma/\nu$, $\beta/\nu$ and $\alpha$ as described in the text. For the exponent $\alpha$, try fitting to a power and a logarithm and compare the results according to the discussion in the text.

10. Consider the table 14.5 which gives the results of the measurements of $L$, $\beta_c(L)$, $\chi_{\text{max}}$, $\beta_c'(L)$ and $c_{\text{max}}$. Make the appropriate fits in order to calculate the exponents $1/\nu$, $\gamma/\nu$, $\alpha/\nu$ and the critical temperature $\beta_c$ as described in the text. For the exponent $\alpha$, try fitting to a power and a logarithm and compare the results according to the discussion in the text.

11. Reproduce the collapse of the curves shown in figures 14.27-14.29. Use the data in the file `all` from the accompanying software. Set the appropriate values to the parameters and calculate the scaling functions $F_{\chi,m,c}$. Vary each parameter separately, so that the collapse becomes not satisfactory and use its variation as an estimate of its error. Determine the range in $x = L^{1/\nu}t$ that gives satisfactory collapse

of the curves. Repeat your calculation by performing measurements for $L = 10, 20$, and using the data for $L = 10, 20, 40, 80, 120$. Compare the new results with the previous ones and comment on the finite size effects.

12. Prove that for every observable $\mathcal{O}$ we have that $\partial\langle\mathcal{O}\rangle/\partial\beta = -\langle E\mathcal{O}\rangle + \langle\mathcal{O}\rangle\langle E\rangle = -\langle(E - \langle E\rangle)(\mathcal{O} - \langle\mathcal{O}\rangle)\rangle$. Using this relation calculate the derivative of the Binder cumulant $D_U$ and prove equation (14.67).

13. Use the maximum of the derivative of the Binder cumulant $D_U$ in order to calculate the critical exponent $1/\nu$ according to the analysis shown in figures 14.23 and 14.25 for the magnetic susceptibility.

14. Show that for a Gaussian distribution $f(x) = a\mathrm{e}^{-x^2/2\sigma^2}$ we have that $\langle x^2\rangle = \sigma^2$ and $\langle x^4\rangle = 3\sigma^4$. Conclude that $1 - \langle x^2\rangle/(3\langle x^4\rangle) = 0$.

15. Consider the distribution given by the probability density distribution
$$f(x) = a\left(\mathrm{e}^{-\frac{(x-m)^2}{2\sigma^2}} + \mathrm{e}^{-\frac{(x+m)^2}{2\sigma^2}}\right).$$
Plot this function and comment on the fact that it looks, qualitatively, like the distribution of the magnetization in the low temperature phase $\beta \gg \beta_c$. Show that $\langle x^4\rangle = m^4 + 6m^2\sigma^2 + 3\sigma^2$ and $\langle x^2\rangle = m^2 + \sigma^2$. Interpret your results, i.e. the meaning of each expectation value. Show that for $\sigma \ll m$ we obtain $U \approx 2/3$. Convince yourself that the approximation used concerns the system in the low temperature phase.

16. Calculate the derivative $\partial U/\partial\beta$ as a function of $\langle em^4\rangle$, $\langle em^2\rangle$, $\langle m^4\rangle$ and $\langle m^2\rangle$. Apply finite size scaling arguments and prove equation (14.120).

17. Use equations (14.131) and $y_t = 1/\nu$, $\gamma = (2y_h - d)/y_t$ in order to prove the other relations in (14.132) and (14.133).

# Bibliography

## [Textbooks]


[1] www.physics.ntua.gr/~konstant/ComputationalPhysics/. The site of this book. The accompanying software and additional material can be found there. You may also find contact information about the author for sending corrections and/or suggestions. Fan mail accepted too!

[2] H. Gould, J. Tobochnik και H. Christian, *Computer Simulation Methods, Application to Physical Systems*, Third Edition, Addison Wesley. A great introductory book in computational physics. Java is the programming language of choice and a complete computing environment is provided for using and creating interacting and visual physics applications. The software is open source and can be downloaded from opensourcephysics.org

[3] R. Landau, M.J. Páez, C.C. Bordeianu, *Computational Physics: Problem Solving with Computers*, Wiley-VCH, 2 ed. (2007)

[4] M. E. J. Newman and G. T. Barkema, *Monte Carlo Methods in Statistical Physics*, Clarendon Press, Oxford. Excellent book for an introductory and intermediate level course in Monte Carlo methods in physics.

[5] B. A. Berg, *Markov Chain Monte Carlo Simulations and Their Statistical Analysis. With Web-Based Fortran Code*, World Scientific, 2004. Graduate level Monte Carlo from a great expert in the field. Covers many advanced Monte Carlo methods.

[6] D.P. Landau and K. Binder, *A Guide to Monte Carlo Simulations in Statistical Physics*, Cambridge University Press, 3rd Edition, 2009.

[7] K. Binder, D.W. Heermann, *Monte Carlo Simulation in Statistical Physics*, Fourth Edition, Springer.

[8] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flanney, *Numerical Recipes*, *The Art of Scientific Computing*, Cambridge University Press, `www.nr.com`. Well, this is *the* handbook of every physicist in numerical methods.

## [Chapter 1]

[9] T.M.R Ellis, I.R. Philips, T.M. Lahey, *Fortran 90 Programming*, Addison-Wesley, 1994

[10] J.C. Adams, W.S. Brainerd, R.A. Hendrickson, R.E. Maine, J.T. Martin, B.T. Smith, *The Fortran 2003 Handbook: The Complete Syntax*, *Features and Procedures*, Springer, 2009

[11] Gnuplot official site `http://gnuplot.info/`

[12] Philipp K. Janert, *Gnuplot in Action: Understanding Data with Graphs*, Manning Publications, 2009.

[13] tcsh homepage: `http://www.tcsh.org/Home`

[14] Paul DuBois, *Using csh & tcsh*, O'Reilly and Associates, 1995, `www.kitebird.com/csh-tcsh-book/`.

[15] Malcolm J. Currie, *C-shell Cookbook*, `http://www.astro.soton.ac.uk/unixtut/sc4.pdf`.

[16] Wiki book: *C Shell Scripting*, `http://en.wikibooks.org/wiki/C_Shell_Scripting`.

[17] Gail Anderson and Paul Anderson, *The Unix C Shell Field Guide*,

## [Chapter 3]

[18] R. M. May, "Simple Mathematical Models with Very Complicated Dynamics", Nature 261 (1976) 459. The first pedagogical and relatively brief introduction to the logistic map and its basic properties.

[19] C. Efthimiou, Introduction to Functional Equations: Theory and Problem-Solving Strategies for Mathematical Competitions and Beyond, MSRI Mathematical Circles Library, 2010. Section 16.7 presents a brief and simple presentation of the mathematical properties of the logistic map.

[20] P. Cvitanović, R. Artuso, R. Mainieri, G. Tanner and G. Vattay, *Chaos: Classical and Quantum*, `ChaosBook.org` (Niels Bohr Institute, Copenhagen 2012). An excellent book on the subject. Can be freely downloaded from the site of the book.

[21] L. Smith, Chaos: A Very Short Introduction, Oxford University Press, 2007.

[22] M. Schroeder, Fractals, Chaos, Power Laws: Minutes from an Infinite Paradise, W.H. Freeman 1991.

[23] S.H. Strogatz, Non Linear Dynamics and Chaos, Addison-Wesley 1994.

[24] Wikipedia: "Chaos Theory", "Logistic Map", "Bifurcation Diagram", "Liapunov Exponents", "Fractal Dimension", "Feigenbaum constants".

[25] Wikipedia: "List of chaotic maps".

[26] Wikipedia: "Newton's method".

[27] M. Jakobson, "Absolutely continuous invariant measures for one-parameter families of one-dimensional maps", Commun. Math. Phys. **81** (1981) 39.

## [Chapter 4]

[28] *Numerical Recipes* [8]. See chapters on the Runge–Kutta methods.

[29] Weisstein, Eric W. *Runge-Kutta Method*, from MathWorld–A Wolfram Web Resource.
http://mathworld.wolfram.com/Runge-KuttaMethod.html.

[30] J. H. E. Cartwright and O. Piro, *The dynamics of Runge-Kutta methods*, Int. J. Bifurcation and Chaos 2, 427-449, 1992.

[31] John H. Mathews, Kurtis Fink, *Numerical Methods Using Matlab*, Prentice Hall, 2003, Chapter 9.

[32] John H. Mathews, *Numerical Analysis - Numerical Methods Project*, `http://math.fullerton.edu/mathews/numerical.html`.

[33] Ian Percival and Derek Richards, *Introduction to Dynamics*, Cambridge University Press, 1982. See also [35].

[34] J. B. McLaughlin, *Period Doubling bifurcations and chaotic motion for a parametrically forced pendulum*, J. Stat. Phys. 24, 375–388 (1981).

## [Chapter 5]

[35] J.V. José, E.J. Saletan, *Classical Dynamics, a Contemporary Approach*, Cambridge University Press, 1998. A great book on Classical Mechanics. You will find a lot of information on non linear dynamical systems exhibiting chaotic behavior. See also the chapters on scattering and planetary motion.

## [Chapter 6]

[36] R.W. Brankin, I. Gladwell, and L.F. Shampine, RKSUITE: a suite of Runge-Kutta codes for the initial value problem for ODEs, Softreport 92-S1, Department of Mathematics, Southern Methodist University, Dallas, Texas, U.S.A, 1992. Available at `www.netlib.org/ode/rksuite` and the accompanying software of the book.

## [Chapter 9]

[37] See the Mathematica Notebooks of Peter West `http://young.physics.ucsc.edu/115/`.

[38] U. Wolff, B. Bunk, F. Knechtli, Computational Physics I, `http://www.physik.hu-berlin.de/com/teachingandseminars/previous_CPI_CPII`.

[39] F. T. Hioe and E. W. Montroll, *Quantum theory of anharmonic oscillators. I. Energy levels of oscillators with positive quartic anharmonicity*, J. Math. Phys. 16, 1945 (1975); `http://dx.doi.org/10.1063/1.522747`

## [Chapter 11]

[40] L. Kadanoff, *Statistical Physics – Statics, Dynamics and Renormalization*, World Scientific. A great book in advanced statistical physics by one of the greatest in the field!

[41] J. Ambjørn, B. Durhuus, T. Jonsson, *Quantum Geometry*, Cambridge Monographs on Mathematical Physics, Cambridge University Press. More in depth discussion of random walks in field theory and statistical mechanics.

[42] C. Itzykson, J.M. Drouffe, *Statistical Field Theory*, Volume 1, Cambridge Monographs on Mathematical Physics, Cambridge University Press. Random walks and Euclidean quantum field theory.

[43] D. E. Knuth, *Seminumerical Algorithms*, Vol. 2 of *The Art of Computer Programming*, Addison-Wesley, 1981.

[44] M. Lüscher, Comput. Phys. Commun. 79 (1994) 100; F. James, Comput. Phys. Commun. 79 (1994) 111; *Erratum* 97 (1996) 357. The code is available at the journal's site `http://cpc.cs.qub.ac.uk/summaries/ACPR_v1_0.html` as well as from CERN at `http://wwwasd.web.cern.ch/wwwasd/cernlib/download/2001_wnt/src/mathlib/gen/v/ranlux.F`.

[45] L. Schrage, it A More Portable Fortran Random Number Generator, ACM Transactions on Mathematical Software, 5 (1979) 132-138; P. Bratley, B.L. Fox and L. Schrage, *A Guide to Simulation*, Springer-Verlag, 1983.

[46] G. Marsaglia and A. Zaman, Ann. Appl. Prob. 1 (1991) 462

[47] B. Li, N. Madras, A.D. Sokal, *Critical Exponents, Hyperscaling and Universal Amplitude Ratios for Two- and Three-Dimensional Self-Avoiding Walks*, J.Statist.Phys. 80 (1995) 661-754, [hep-lat/9409003]; G. Slade, *The self-avoiding walk: A brief survey*, Surveys in Stochastic Processes, pp. 181-199, eds. J. Blath, P. Imkeller, S. Roelly, European Mathematical Society, Zurich, (2011), `http://www.math.ubc.ca/~slade/spa_proceedings.pdf`.

## [Chapter 12]

[48] J. J. Binney, N. J. Dowrick, A. J. Fisher and M. E. J. Newman, *The Theory of Critical Phenomena*, Oxford. A simple introduction to critical phenomena and the renormalization group.

[49] R. K. Pathria, *Statistical Mechanics*, Pergamon. A classic in statistical physics.

[50] F. Mandl, *Statistical Physics*,Wiley; 2 edition (May 12, 1988)

[51] R. J. Baxter, Exactly Solved Models in Statistical Mechanics, Dover Publications 2008.

## [Chapter 13]

[52] E. Ising, *Beitrag zur Theorie des Ferromagnetizmus*, Z. Phys. **31** (1925) 253–258.

[53] L. Onsager, *Crystal Statistics. I. A Two–Dimensional Model with an Order–Disorder Transition*, Phys. Rev. **65** (1944) 117–119.

[54] K. Huang, *Statistical Mechanics*, John Wiley & Sons, New York, ed. 1987. A detailed presentation of the Onsager solution.

[55] C.N. Yang, Phys. Rev. **85** (1952) 809.

[56] Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H. and Teller, E., J. Chem. Phys. **21** (1953) 1087.

[57] Nightingale, M. P. and Blöte, H. W. J., *Phys. Rev. Lett.* **76** (1996) 4548.

[58] Müller-Krumbhaar, H and Binder, K., *J. Stat. Phys.* **8** (1973) 1.

[59] Efron, Bradley *SIAM Review* **21** (1979) 460; *Ann. Statist.* **7** (1979) 1;Efron, B and Tibshirani, R. *Statistical Science* **1** (1986) 54. Freely available from `projecteuclid.org`.

## [Chapter 14]

[60] Swendsen, R. H. and Wang J.-S., *Phys. Rev. Lett.* **58** (1987) 86

[61] U. Wolff, Phys. Rev. Lett. **62** (1989) 361

[62] A. Pelisseto and E. Vicari, "*Critical Phenomena and Renormalization–Group Theory*", Phys. Reports 368 (2002) 549.

[63] F. Y. Wu, *The Potts Model*, Rev. Mod. Phys. **54** (1982) 235

[64] Coddington, P.D. and Baillie, *Phys. Rev. Lett.* **68** (1992) 962

[65] Rieger, H., *Phys. Rev. B* **52** (1995) 6659

[66] Newman E. J. and Barkema, G. T. *Phys. Rev. E* **53** (1996) 393

[67] A.E. Ferdinand and M.E. Fisher, *Phys. Rev.* **185** (1969) 832; N.Sh. Izmailian, C.-K. Hu, *Phys. Rev.* **E 65** (2002) 036103; J. Salas, *J. Phys.* **A 34** (2001) 1311; W. Janke and R. Kenna, *Nucl. Phys. (Proc. Suppl.)* **106** (2002) 929.

[68] J. Ambjørn, K.N. Anagnostopoulos, *Nucl. Phys.* **B 497** (1997) 445.

[69] K. Binder, *Phys. Rev. Lett.* **47** (1981) 693.

[70] K. Binder, *Z. Phys.* **B 43** (1981) 119; G. Kamieniarz and H.W.J. Blöte, *J. Phys. A* **26** (1993) 201.

[71] J. Cardy, *Scaling and Renormalization in Statistical Physics*, Cambridge University Press, 1st Edition, 1996.

[72] A. M. Ferrenberg and D. P. Landau, *Phys. Rev.* **B44** (1991) 5081.

[73] M. S. S. Challa, D. P. Landau and K. Binder, *Phys. Rev.* **B34** (1986) 1841.

[74] H.E. Stanley, "*Introduction to Phase Transitions and Critical Phenomena*", Oxford 1971.

[75] R. Creswick and S.-Y. Kim, *J. Phys. A: Math.Gen.* **30** (1997) 8785

[76] C. Holm and W. Janke, *Phys. Rev.* **B 48** (1993) 932, [arXiv:hep-lat/9301002].

[77] M. Hasenbusch and S. Meyer, *Phys. Lett.* **B 241** (1990) 238

[78] M. Kolesik and M. Suzuki arXiv:cond-mat/9411109

[79] M. Kolesik and M. Suzuki, *Physica* **A 216** (1995) 469.

# Index


$\beta_c$, 499, 574
$\chi^2$/dof, 548
. (current directory), 5
.. (parent directory), 5
; (separate commands), 10
$PATH, 11
- (switch, options), 11
/dev/random, 444, 457, 555
/dev/urandom, 444, 457, 555
/, 5
< (redirection), 13
>> (redirection), 13
>& (redirection), 13
> (redirection), 13
NF, 20
#!, 592
$path, 11
& (background a process), 22
a.out, 83
ar, 380
awk, 20, 66
    BEGIN, 20, 227
    END, 20, 470
    NR, 20, 469
    $1, $2, ..., 20
    script, 470
cat, 17, 66
cd, 6
chmod, 8
cp, 8
date, 66
echo, 66
emacs, 21
gfortran, 36, 82
    link library, 380
grep, 19
head, 17
info, 14
less, 17
ls, 7
make, 526
man, 14
mkdir, 6
mv, 9
pwd, 6
rk2.csh, 250
rmdir, 6, 10
rm, 9
setenv, 11
set, 11
sort, 18
stderr, 12
stdin, 12
stdout, 12
tail, 17
time, 526
whatis, 15
where, 15
which, 12, 15
| (piping), 14, 558

absolute path, 3
acceptance ratio, 493, 502
    average, 508, 564
anharmonic oscillator, 375, 431
annihilation operator, 374
attractor, 148, 222

autocorrelation
  critical slowing down, 535
  dynamical exponent $z$, 535
  function, 530, 543
  independent measurement, 532, 539
  time, 531, 549
    subdominant, 549
  time, integrated, 533

basin of attraction, 158, 182, 228
bifurcation, 148, 154, 164
Binder cumulant, 627
Boltzmann constant, 477
Boltzmann distribution, 477
bootstrap, 542, 555, 561
boundary conditions
  fixed, 504
  free, 504
  helical, 505, 506
  periodic, 496, 505
  toroidal, 496, 505
boundary value problem, 334

canonical ensemble, 476, 477
chaos, 145, 150, 226
  period doubling, 226
  pseudorandom numbers, 444
circle map, 186
cluster, 576, 579
cluster algorithms, 575
cluster seed, 576
cobweb plot, 152
cold state, 506, 527
collapse, 617
command completion, 16
command substitution, 65
completion
  command, 16
  filename, 16
conformable arrays, 53, 276
conjugate thermodynamic quantities, 484
continuum limit, 487
correlation function, 485, 500
correlation length, 486, 500, 573
Coulomb's law, 311
Courant parameter, 358
CPU time, 526
creation operator, 374
critical exponents, 465, 500, 573, 641
  $\alpha$, 500, 574, 637, 641
  $\beta$, 500, 641
  $\beta$ , 574
  $\delta$, 500, 641
  $\eta$, 487, 500, 641
  $\gamma$, 500, 574, 641
  $\nu$, 465, 573, 641
  $y_h$, 636, 641
  $y_t$, 636, 641
  $z$, 535, 575, 641
critical slowing down, 535, 575
cross section, 258
  differential, 258, 260
  total, 258
cumulant, 627
cumulative distribution function, 451
Curie temperature, 499
current directory, 4

density of states, 480
dependencies, 527
derivative
  numerical, 164, 357
derivative, numerical, 426
detailed balance, 502
detailed balance condition, 491
diagonalization, 379
diffusion, 363
  equation, 353
  kernel, 353
Dirac delta function, 353

directory, 4
  home, 3, 5
  parent, 4
Dirichlet boundary condition, 356
disordered phase, 499
double well, 390, 413
DSYEV, 376
Duffing map, 188
dynamic exponent $z$, 575, 602
dynamic memory allocation, 270

eccentricity, 254
eigenstate, 374, 402
eigenvalues, 379
eigenvectors, 379
electric equipotential surfaces, 312
electric field, 311, 334
electric field lines, 312
electric potential, 312, 334
Emacs, 21
  abort command, 74, 76
  auto completion, 33
  commands, 74
  Ctrl key, `C-`, 23
  cut/paste, 27, 75
  edit a buffer, 26
  frames, 29
  help, 32, 74
  info, 32, 76
  kill a buffer, 30
  mark, 26
  Meta key, `M-`, 23
  minibuffer, 76
  minibuffer, `M-x`, 23
  modes, 31
    LaTeX, 31
    auto fill, 32
    C, 31
    font lock (coloring), 32
    Fortran, 31
    overwrite, 32
  point, 26
  read a file, 26, 30, 74
  recover a buffer, 30
  recover file, 74
  region, 26
  replace, 75
  save a buffer, 26, 30, 74
  search, 74
  spelling, 76
  undo, 27, 74
  window, split, 29, 75
  windows, 29, 75
energy spectrum, 375
entropy, 176, 479
ergodicity, 491, 503
error
  binning, 541
  blocking, 541
  bootstrap, 542, 555, 561
    binning, 561
  error of the mean, 538
  integration, 114
  jackknife, 540, 551
  statistical, 470, 538
  systematic, 536
estimator, 205, 487
Euler method, 193
Euler-Verlet method, 194, 229
expectation value, 387, 403, 426, 478

Feigenbaum constant, 156
FIFO, 583
file
  owner, 7
  permissions, 7
filename completion, 16
filesystem, 3
finite size effects, 574
finite size scaling, 638
first order phase transition, 482
fit, 170, 470

$\chi^2$/dof, 548
variance of residuals, 549
fluctuations, 483
focus,foci, 254
foreach, 386, 591
Fortran
RANDOM_SEED, 457
RANLUX, 462, 587
allocatable, 271, 516
ALLOCATE, 271, 519, 555
ALLOCATED, 72
arrays, 50
bound upper/lower, 52
conformable, 53, 276
constructor, 53
dimension, 51, 52
DOT_PRODUCT, 55
extent, 52
LBOUND, 55
MATMUL, 55
MAXVAL, 55
MINVAL, 55
PRODUCT, 55
range, 51
rank, 52
RESHAPE, 56
scalar, 52
section, 276
shape, 52, 271
size, 52
SUM, 55
TRANSPOSE, 55
UBOUND, 55
vector operations, 276
call, 42
CMPLX, 71
column limits, 35
comments, 35
common blocks, 47, 218
comparison operators, 45
compile, 36
complex, 45
CONJG, 71
CONTAINS, 546
continuation of lines, 35
DATE_AND_TIME, 71
DBLE, 71
DEALLOCATE, 273
dimension, 38
do, 38
do while, 82, 109
DOT_PRODUCT, 73
DSYEV, 376
elemental, 53
EPSILON, 45, 71
FDATE, 525
FILE, 41
FORALL, 56
format, 49, 90
function, 44
GETARG, 430, 522
GETENV, 525
GETLOG, 525
GETOPT, 522, 546, 551
Hello World!, 35
HUGE, 45, 71
IARGC, 430
IF, 43, 45, 81
implicit, 37, 40
implied do loops, 55
INT, 71
INTERFACE, 276
intrinsic functions, 71
labeled statement, 50, 90
main program, 36
MATMUL, 73
MAXVAL, 73
MINVAL, 73
MOD, 72
module, 515, 544, 555
OPEN, 40, 82
options, 522

PACK, 73
parameter, 41
PRECISION, 45
PRODUCT, 73
RANDOM_NUMBER, 72, 456
RANDOM_SEED, 72
RANGE, 45
READ, 40
END (end of file), 521
real, 45
real, accuracy, 111
RENAME, 521
RESHAPE, 73
rksuite, 286
SAVE, 456, 515, 544
scalar, 52
SELECT, 467, 520, 522
SHAPE, 73
SIZE, 73
SLEEP, 468
STOP, 43, 81
string comparison, 72
subroutine, 41
SUM, 73, 544, 555
TINY, 45, 72
TRANSPOSE, 73
TRIM, 45, 72, 524
UBOUND, 73
unit, 40, 82
WHERE, 56
WRITE, 40
free energy, 479

Gauss map, 186
Gauss's law, 334
Gauss-Seidel overrelaxation, 347
Gaussian distribution, 453
Gibbs, 477
Gnuplot, 57
<, 60, 249, 361, 424
1/0 (undefined number), 249
animation, 86, 102
atan2, 86
comment, 59
fit, 170, 470, 548
functions, 203
hidden3d, 61
load, 87, 120
log plots, 170
parametric plot, 62
plot, 59
plot 3d, 61, 102, 342, 361
plot command output, 60, 249, 361, 424
plot expressions, 59, 86, 249
pm3d, 61
replot, 59, 61
reread, 249
save plots, 61
splot, 61, 102, 342, 361
using, 59, 86
variables, 96, 203
with, 59
ground state, 402, 476, 498
ground state energy, 476

Hénon map, 187
hard sphere, 257
harmonic oscillator, 373, 431
heat conduction, 355
heat reservoir, 476
Heisenberg's uncertainty principle, 402
Heisenberg's uncertainty relation, 388, 426
helical boundary conditions, 505, 506
high temperature phase, 499
histogram, 449
home directory, 3, 5
hot state, 506, 527
hyperscaling, 642
hyperscaling relations, 635, 637, 641

impact parameter, 259, 260
importance sampling, 489
independent measurement, 532, 539
initial state, 527
internal energy, 478
Ising model
  $Z_2$ symmetry, 498, 501
  $\beta_c$, 499
  energy, 508
  ferromagnetic, 498
  Hamiltonian, 498, 501
  magnetization, 508
  partition function, 498, 502

jackknife, 540, 551
Jacobi overrelaxation, 347

Kepler's law, 254

Lapack, 376
Laplace equation, 334
lattice
  constant, 334, 496
  triangular, 568
leapfrog method, 233
Lennard-Jones potential, 435
Liapunov exponent, 167
libblas, 380
liblapack, 380
LIFO, 583
linear coupling, 484
logistic map, 146
  $2^n$ cycles, 149
  attractor, 148
  bifurcation, 148, 154, 164
  cobweb plot, 152
  entropy, 176
  fixed points, 147
    stability, 148
  onset of chaos, 150, 175
  special solutions, 147
  strong chaos, 174
  transient behavior, 154
  weak chaos, 174
low temperature phase, 499

magnetic susceptibility, 485, 509
  scaling, 574
magnetization, 484, 508
  scaling, 574
  staggered, 569
magnetized, 499
Makefile, 526
man pages, 14
Markov chain, 490
Markov process, 490
Marsaglia and Zaman, 445
master equation, 476
memory
  allocation, dynamic, 270
  allocation, static, 270
  leak, 273
Metropolis algorithm, 503
minibuffer, 23
Monte Carlo
  cold state, 506, 527
  hot state, 506, 527
  initial state, 506, 527
  simulation, 490, 492
  sweep, 530, 601, 603, 642
mouse map, 186

Netlib, 286, 376
  Blas, 380
  Lapack, 376
    DSYEV, 376
  liblapack, 380
  rksuite, 286
Newton's law of gravity, 253
Newton-Raphson method, 157, 161
NRRW (non reversal random walker), 465
numerical

derivative, 164, 357, 426
integration, 419

observable, 477
Onsager, 496
exponents, 500
Onsager critical exponents, 574
options, 11, 522
order parameter, 501
ordered phase, 499
overflow, 446
overlap, 488
overrelaxation, 336, 347

parent directory, 4
parity, 405
partition function, 477
Ising model, 498
path
absolute, 3
command, 11
file, 3
relative, 4
period, 445, 456
period doubling, 150
periodic boundary conditions, 496, 505
phase transition
1st order, 482, 566
2nd order, 499
continuous, 499
piping, 14
Poincaré diagram, 226
Poisson equation, 342
pseudocritical region, 574, 613
pseudocritical temperature, 574
pseudorandom, 444

queue, 583

random
`RANLUX`, 462, 587
`RLUXGO`, 461
`RLUXIN`, 461
`RLUXUT`, 461
`drandom()`, 446
`gaussran()`, 454
`naiveran()`, 445
Cauchy distribution, 452
chaos, 444
correlations, 447
Gaussian distribution, 453
generator, 444
Marsaglia and Zaman, 445
modulo generator, 444
multiply-with-carry, 456
non uniform, 451
period, 445, 456, 460
pseudorandom, 444
Ranlux, 460
save state, 456, 461
Schrage, 446
seed, 457, 520
uniform, 449
urandom, 444, 457, 555
random walk, 353, 463
NRRW, 465
SAW, 465
Ranlux, 460
`RANLUX`, 462, 587
`RLUXGO`, 461, 517
`RLUXIN`, 461, 517
`RLUXUT`, 461, 521
`ranlux_level`, 461, 516
save state, 461
redirection, 13
relative path, 4
relativity
special, 297
reservoir, heat, 476
return probability, 355
rksuite, 286
root, 4

Runge-Kutta method, 204, 233, 242, 407, 418
    adaptive stepsize, 286
Rutherford scattering, 260
RW (random walker), 463

sample, 487
sampling, 488
    importance, 489
    simple, 488
SAW (self avoiding walk), 465
scale invariance, 501
scaling, 500
    collapse, 617
    exponents, 500
    factor, 636
    hypothesis, 632, 636
scattering, 257, 262
    rate, 258
    Rutherford, 260
Schrödinger equation, 401
Schrage, 446
second order phase transition, 499
seed, 457, 520
seed of cluster, 576
selection probability, 493, 502
shell
    argv, 66
    array variable, 64
    arrays, 66
    command substitution, 65
    foreach, 66
    here document, 63, 66
    if, 66
    input $<, 66
    script, 62, 66
    set, 64
    tcsh, 62
    variable, 64
shell script, 590
Simpson's rule, 419
sine map, 185
SOR, successive overrelaxation, 336, 348
specific heat, 478, 509
    scaling, 574
spectral dimension, 355
spin
    configuration, 498
spin cluster, 576, 579
splinter, 147
stack, 583
staggered, 569
standard deviation, 450
standard error, 12
standard input, 12
standard output, 12
statistical physics, 476
subdirectory, 4
successive overrelaxation, 336
susceptibility, magnetic, 485
sweep, 530, 601, 603, 642
symmetry breaking, 501

tcsh, 591
temperature, 476
tent map, 186
thermal conductivity, 355
thermal diffusivity, 355
thermalization, 527
    discard, 594
    time, 490
thermodynamic limit, 478
third law of thermodynamics, 479
timing jobs, 526
Tinkerbell map, 189
toroidal boundary conditions, 496, 505
transient behavior, 154, 221
transition probability, 476, 490, 492
transition rates, 476
tunneling, 390, 414

turning point, 416

universality, 487, 501, 573
    class, 500
universality class, 573
user interface, 81

variables
    environment, 11
    shell, 11
variance, 450

wave function, 374
weights, statistical, 476
Wolff cluster algorithm, 576, 579
working directory, 4

This book is an introduction to the computational methods used in physics, but also in other scientific fields. It is addressed to an audience that has already been exposed to the introductory level of college physics, usually taught during the first two years of an undergraduate program in science and engineering. It assumes no prior knowledge of numerical analysis, programming or computers and teaches whatever is necessary for the solution of the problems addressed in the text.

The book starts with very simple problems in particle motion and ends with an in-depth discussion of advanced techniques used in Monte Carlo simulations in statistical mechanics. The level of instruction rises slowly, while discussing problems like the diffusion equation, electrostatics on the plane, quantum mechanics and random walks. All the material can be taught in two semesters, but a selection of topics can form the material of a one semester course. The book aims to provide the students with the background and the experience needed in order to advance to high performance computing projects in science and engineering. It puts emphasis on hands–on programming of numerical code but also on the production, analysis and interpretation of data. But it also tries to keep the students motivated by considering interesting applications in physics, like chaos, quantum mechanics, special relativity and the physics of phase transitions.

The book and the accompanying software are given under a Creative Commons License/GNU public License as a service to the community. It can be used freely as a whole, or any part of it, in any form, by anyone. There is no official distribution of hard copies, but you can use the printing service of your preference in order produce any number of copies you need for you and/or your students. The ebook can be read in most electronic devices like your PC, tablet or favorite ebook reader and it is freely available from the book's website.

**Konstantinos Anagnostopoulos** was educated at the University of Athens and at Syracuse University where he obtained his Ph.D. in theoretical physics in 1993. He has been a researcher at the Institute for Fundamental Theory at the University of Florida, Gainesville and at the Niels Bohr Institute at the University of Copenhagen from 1993 to 1999. He held a visiting professor position at the University of Crete from 1999-2004, and in 2004 he moved to Athens where he has been a member of the faculty of the Physics Department of the National Technical University of Athens (NTUA) ever since. His area of expertise is computational physics with an emphasis in theoretical high energy physics, string theory and quantum gravity. He is a world expert in Monte Carlo simulations of such systems, and he has participated in projects run onto the world's largest supercomputers. At NTUA he founded three undergraduate computational physics and scientific computing courses at the senior and sophomore level, which he has been teaching since 2004. The audience of those courses are students interested in theoretical physics, but also students majoring in a wider area of applied mathematics, physics and engineering.